# An Overview of OpenMP

**Ruud van der Pas**

**Senior Staff Engineer**
**Oracle Solaris Studio**
**Oracle**
**Menlo Park, CA, USA**

**IWOMP 2010**
**CCS, University of Tsukuba**
**Tsukuba, Japan**
**June 14-16, 2010**

# Outline

- ***Getting Started with OpenMP***
- ***Using OpenMP***
- ***Tasking in OpenMP***
- ***Oracle Solaris Studio support for OpenMP***

# *Getting Started with OpenMP*

**http://www.openmp.org**

**http://www.compunity.org**


OpenMP.org
http://openmp.org/wp/
http://www.openmp.org
OpenMP
THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING
Subscribe to the News Feed
»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum
Events
»IWOMP 2010 - 6th International Workshop on OpenMP, June 14-16, 2010, Tsukuba, Japan
Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org
Search OpenMP.org
Search
Archives
May 2010
April 2010
June 2009
April 2009
March 2009
OpenMP News
»SPEC Looking For A Few Good Applications
SPEC, the Standard Performance Evaluation Corporation, is looking for realistic OpenMP applications to include in the next version of the SPEC CPU and SPEC OMP benchmark suites.
SPEC is sponsoring a search program, and for each step of the process that a submission passes, SPEC will compensate the Program Submitter (in recognition of the Submitter's effort and skill). A submission that passes all of the steps and is included in the next SPEC CPU benchmark suite will receive $5000 US overall and a license for the new benchmark suite when released. Details on the Benchmark Search Program at: http://www.spec.org/cpuv6/.
Posted on May 20, 2010
»IWOMP 2010: International Workshop on OpenMP
6th International Workshop on OpenMP, June 14-16, 2010, Tsukuba, Japan
"Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More"
The International Workshop on OpenMP is an annual series of workshops dedicated to the promotion and advancement of all aspects focusing on parallel programming with OpenMP. OpenMP is now a major programming model for shared memory systems from multi-core machines to large scale servers. Recently, new ideas and challenges are proposed to extend OpenMP framework for adopting accelerators and also exploiting parallelism beyond loop levels. The workshop serves as a forum to present the latest research ideas and results related to this shared memory programming model. It also offers the opportunity to interact with OpenMP users, developers and the people working on the next release of the standard. The 2010 International Workshop on OpenMP IWOMP 2010 will be held in the high-tech city of Tsukuba, Japan.
The workshop IWOMP 2010 will be a three-day event. In the first day, tutorials are provided for focusing on topics of interest to current and prospective OpenMP developers, suitable for both
The OpenMP API
supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»Read about OpenMP.org
Get
»OpenMP specs
Use
»OpenMP Compilers
Learn
Using OpenMP
»Using OpenMP – the book

# Shameless Plug - "Using OpenMP"

***"Using OpenMP"***

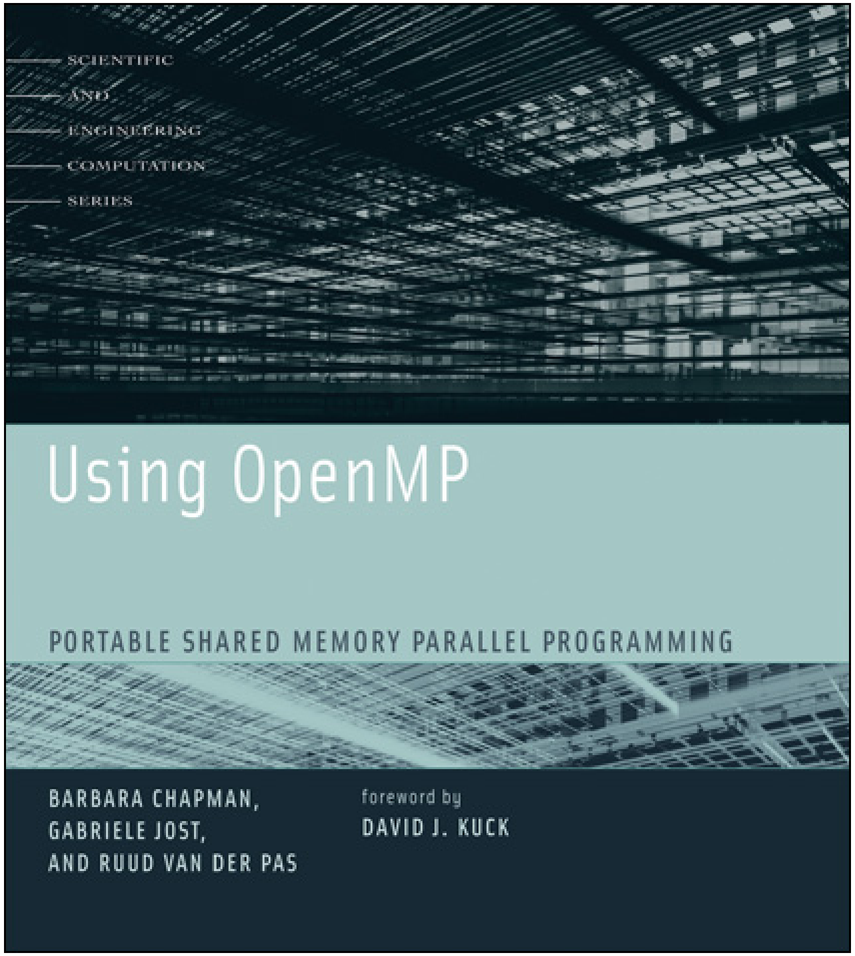
***Portable Shared Memory Parallel Programming***

***Chapman, Jost, van der Pas***

**MIT Press, 2008**

**ISBN-10: 0-262-53302-2**
**ISBN-13: 978-0-262-53302-7**

**List price: 35 $US**



**All 41 examples are available NOW!**

**As well as a forum on http://www.openmp.org**

# OpenMP News

**»Download Book Examples and Discuss**

Ruud van der Pas, one of the authors of the book ***Using OpenMP - - Portable Shared Memory Parallel Programming*** by Chapman, Jost, and van der Pas, has made 41 of the examples in the book available for download and your use.

These source examples are available as a free download **»here** (a zip file) under the BSD license. Each source comes with a copy of the license. Please do not remove this.

You are encouraged to try out these examples and perhaps use them as a starting

***Download the examples and discuss in forum:***

**http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss**

book.

To make things easier, each source directory has a make file called "Makefile". This file can be used to build and run the examples in the specific directory. Before you do so, you need to activate the appropriate include line in file Makefile. There are include files for several compilers and Unix based Operating Systems (Linux, Solaris and Mac OS to precise).

These files have been put together on a best effort basis. The User's Guide that is bundled with the examples explains this in more detail.

Also, we have created a new forum, **»Using OpenMP - The Book and Examples**, for discussion and feedback.

## The OpenMP API

**The OpenMP API** supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

**»OpenMP Compilers**

## Learn

# What is OpenMP?

- ***De-facto standard Application Programming Interface (API) to write <u>shared memory parallel applications</u> in C, C++, and Fortran***
- ***Consists of:***
  - ***Compiler directives***
  - ***Run time routines***
  - ***Environment variables***
- ***Specif cation maintained by the OpenMP Architecture Review Board (http://www.openmp.org)***
- ***Version 3.0 has been released May 2008***

# When to consider OpenMP?

- ***Using an automatically parallelizing compiler:***
  - ***It can not find the parallelism***
    - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize or not***
  - ***The granularity is not high enough***
    - ✔ ***The compiler lacks information to parallelize at the highest possible level***
- ***Not using an automatically parallelizing compiler:***
  - ***No choice than doing it yourself***

# Advantages of OpenMP

- ***Good performance and scalability***
  - ***If you do it right ....***
- ***De-facto and mature standard***
- ***An OpenMP program is portable***
  - ***Supported by a large number of compilers***
- ***Requires little programming effort***
- ***Allows the program to be parallelized incrementally***

# OpenMP and Multicore

**_OpenMP is ideally suited for multicore architectures_**

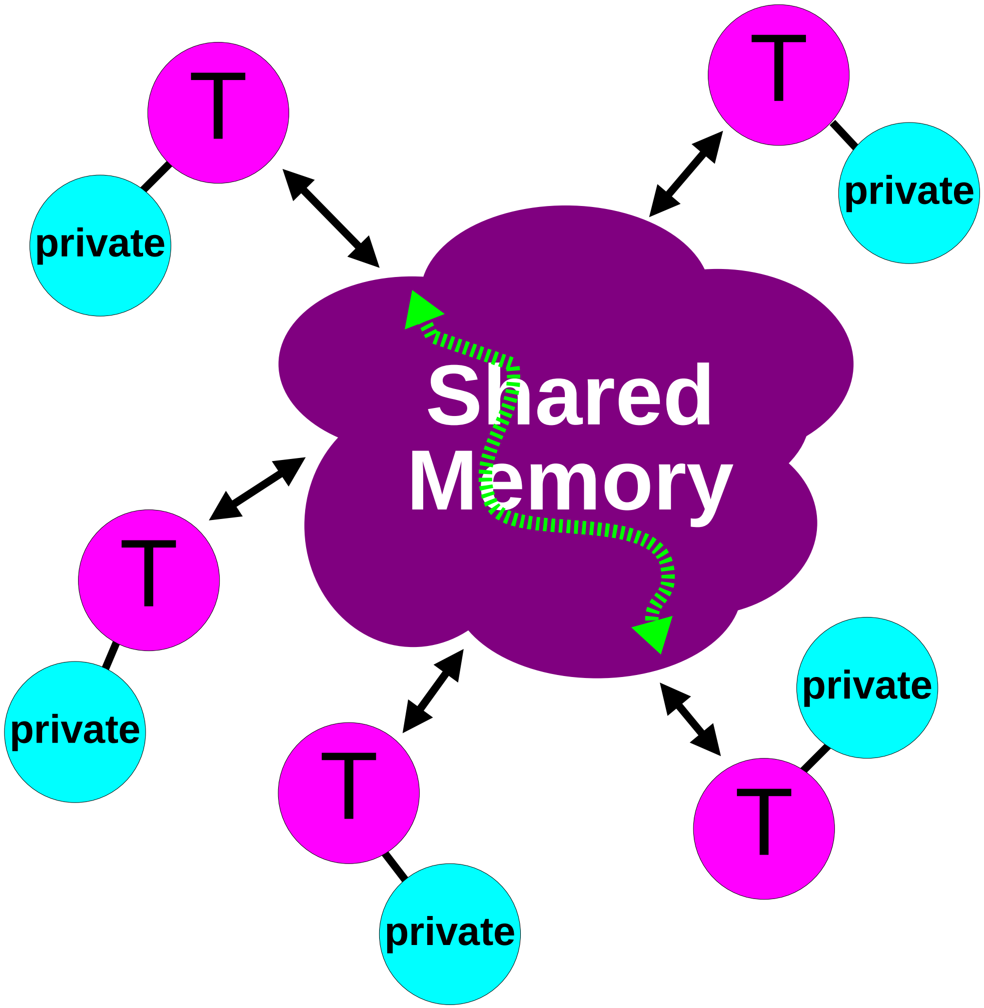
***Memory and threading model map naturally***

***Lightweight***

***Mature***

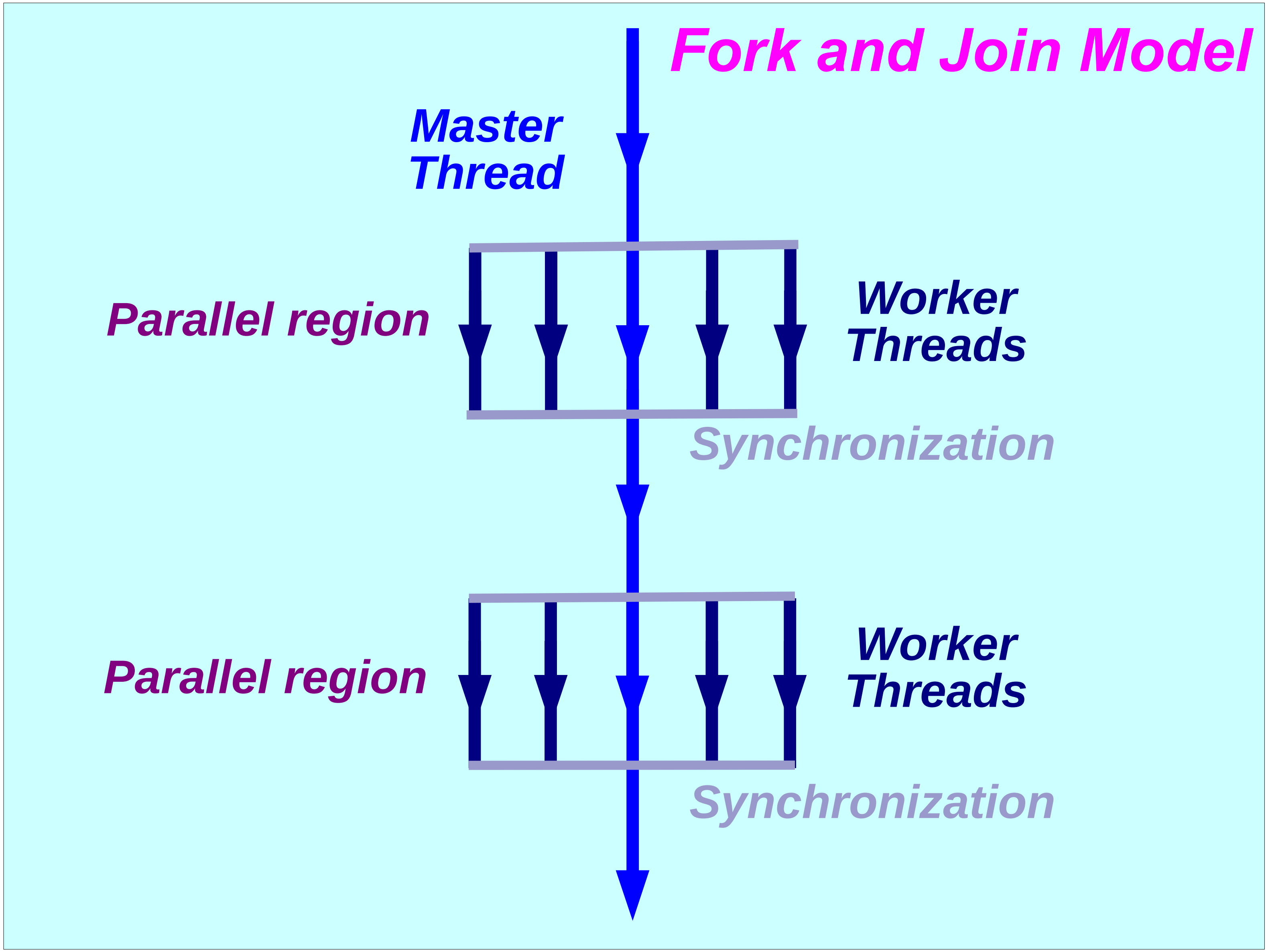
***Widely available and used***

# The OpenMP Memory Model




- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

# The OpenMP Execution Model



# Data-sharing Attributes

- *In an OpenMP program, data needs to be "labeled"*
- *Essentially there are two basic types:*
  - *Shared - There is only one instance of the data*
    - ✔ *All threads can read and write the data simultaneously, unless protected through a specif c OpenMP construct*
    - ✔ *All changes made are visible to all threads*
      - ◆ *But not necessarily immediately, unless enforced ......*
  - *Private - Each thread has a copy of the data*
    - ✔ *No other thread can access this data*
    - ✔ *Changes only visible to the thread owning the data*

# The private and shared clauses

**private (list)**

- *No storage association with original object*
- *All references are to the local object*
- *Values are undefined on entry and exit*

**shared (list)**

- *Data is accessible by all threads in the team*
- *All threads access the same address space*

# An OpenMP example

## For-loop with independent iterations

```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

## For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

```
$ cc -xopenmp source.c
$ export OMP_NUM_THREADS=5
$ ./a.out
```

# Example Parallel Execution

| Thread 0 | Thread 1 | Thread 2 | Thread 3 | Thread 4 |
|---|---|---|---|---|
| i=0-199 | i=200-399 | i=400-599 | i=600-799 | i=800-999 |
| `a[i]` | `a[i]` | `a[i]` | `a[i]` | `a[i]` |
| + | + | + | + | + |
| `b[i]` | `b[i]` | `b[i]` | `b[i]` | `b[i]` |
| = | = | = | = | = |
| `c[i]` | `c[i]` | `c[i]` | `c[i]` | `c[i]` |

# Defining Parallelism in OpenMP

- ***OpenMP Team := Master + Workers***
- ***A <u>Parallel Region</u> is a block of code executed by all threads simultaneously***
  - ***The master thread always has thread ID 0***
  - ***Thread adjustment (if enabled) is only done before entering a parallel region***
  - ***Parallel regions can be nested, but support for this is implementation dependent***
  - ***An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially***
- ***A <u>work-sharing construct</u> divides the execution of the enclosed code region among the members of the team; in other words: they split the work***

# Directive format

- ❑ ***C: directives are case sensitive***
  - *Syntax:* **#pragma omp directive [clause [clause] ...]**
- ❑ ***Continuation: use \ in pragma***
- ❑ ***Conditional compilation:*** **_OPENMP** ***macro is set***

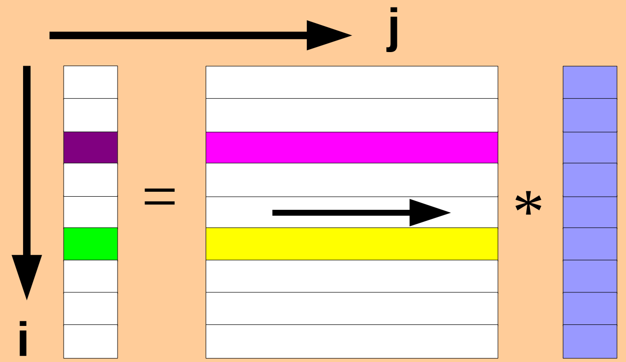
- ❑ ***Fortran: directives are case insensitive***
  - *Syntax:* **sentinel directive [clause [[,] clause]...]**
  - ***The sentinel is one of the following:***
    - ✔ ***!$OMP or C$OMP or *$OMP*** ***(fixed format)***
    - ✔ ***!$OMP*** ***(free format)***
- ❑ ***Continuation: follows the language syntax***
- ❑ ***Conditional compilation:*** **!$** ***or*** **C$ -> 2 spaces**

# OpenMP clauses

- ***Many OpenMP directives support clauses***
  - ***These clauses are used to provide additional information with the directive***
- ***For example, private(a) is a clause to the "for" directive:***
  - **#pragma omp for private(a)**
- ***The specific clause(s) that can be used, depend on the directive***

# Example 2 - Matrix times vector

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```




**TID = 0**

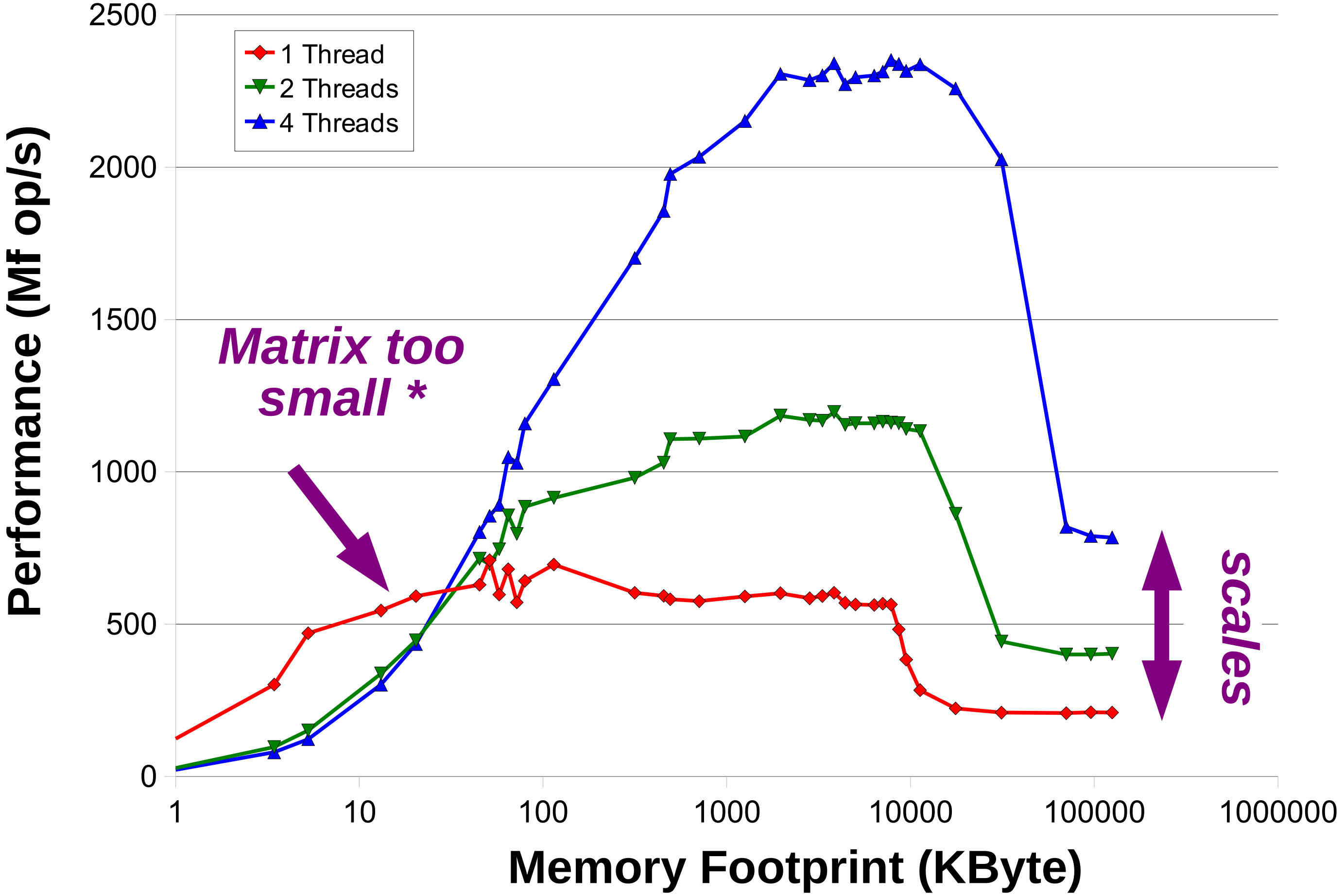
```
for (i=0,1,2,3,4)
 i = 0
sum =   b[i=0][j]*c[j]
  a[0] = sum
 i = 1
sum =   b[i=1][j]*c[j]
  a[1] = sum
```

**TID = 1**

```
for (i=5,6,7,8,9)
 i = 5
sum =   b[i=5][j]*c[j]
  a[5] = sum
 i = 6
sum =   b[i=6][j]*c[j]
  a[6] = sum
```

*... etc ...*

# OpenMP Performance Example




***) With the IF-clause in OpenMP this performance degradation can be avoided**

# The if clause

**if (scalar expression)**

- ✔ *Only execute in parallel if expression evaluates to true*
- ✔ *Otherwise, execute serially*

```
#pragma omp parallel if (n > some_threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

# Barrier/1

***Suppose we run each of these two loops in parallel over i:***

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

***This may give us a wrong answer (one day)***

***Why ?***

# Barrier/2

*We need to have updated all of a[ ] first, before using a[ ] **

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

**wait !**

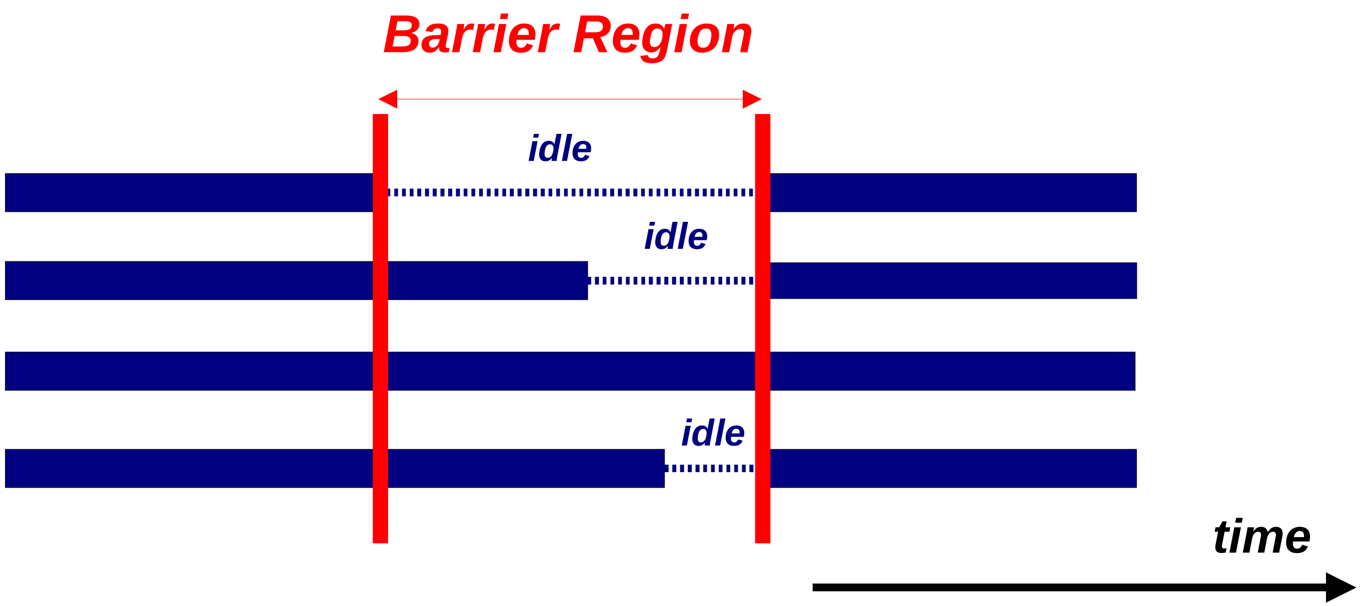
**barrier**

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

***All threads wait at the barrier point and only continue when all threads have reached the barrier point***

****) If there is the guarantee that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case***

# Barrier/3




## *Barrier syntax in OpenMP:*

```
#pragma omp barrier
```

```
!$omp barrier
```

# When to use barriers ?

- ***If data is updated asynchronously and data integrity is at risk***
- ***Examples:***
  - ***Between parts in the code that read and write the same section of memory***
  - ***After one timestep/iteration in a solver***
- ***Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors***
- ***Therefore, use them with care***

# The nowait clause

- *To minimize synchronization, some OpenMP directives/pragmas support the optional* **nowait** *clause*
- *If present, threads do not synchronize/wait at the end of that particular construct*
- *In Fortran the* **nowait** *clause is appended at the closing part of the construct*
- *In C, it is one of the clauses on the pragma*

```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
        :
        :
!$omp end do nowait
```

# A more elaborate example

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];


#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

         ....
#pragma omp barrier
    scale = sum(a,0,n) + sum(z,0,n) + f;
         ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

**parallel loop** (work is distributed)

**parallel loop** (work is distributed)

**synchronization**

Statement is executed by all threads

**parallel region**

# Components of OpenMP

## Directives

- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

## Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Wallclock timer
- Locking

## Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

# The Parallel Region

***A parallel region is a block of code executed by multiple threads simultaneously***

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this code is executed in parallel"

} (implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]

    "this code is executed in parallel"

!$omp end parallel (implied barrier)
```

# The Worksharing Constructs

## *The OpenMP worksharing constructs*

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- *The work is distributed over the threads*
- *Must be enclosed in a parallel region*
- *Must be encountered by all threads in the team, or none at all*
- *No implied barrier on entry; implied barrier on exit (unless nowait is specif ed)*
- *A work-sharing construct does not launch any new threads*

# The Workshare construct

*Fortran has a fourth worksharing construct:*

```
!$OMP WORKSHARE

    <array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

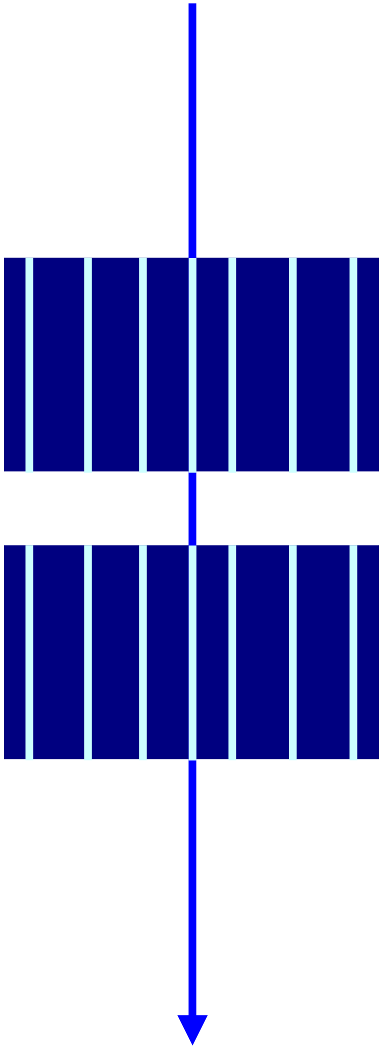
*Example:*

```
!$OMP WORKSHARE
    A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

# The omp for directive - Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

*(implied barrier)*



# C++: Random Access Iterator Loops

***Parallelization of random access iterator loops is supported***

```
void iterator_example()
{
  std::vector vec(23);
  std::vector::iterator it;

  #pragma omp for default(none)shared(vec)
  for (it = vec.begin(); it < vec.end(); it++)
  {
    // do work with *it //
  }
}
```

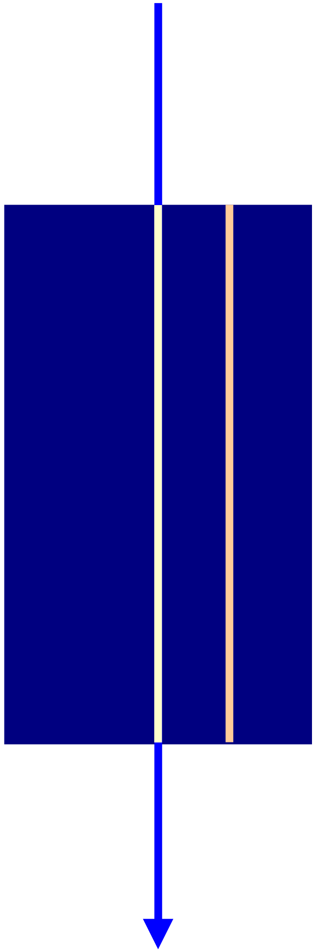
# Loop Collapse

- *Allows parallelization of perfectly nested loops without using nested parallelism*
- **collapse** *clause on for/do loop indicates how many loops should be collapsed*
- *Compiler forms a single loop and then parallelizes this*

```
!$omp parallel do collapse(2) ...
  do i = il, iu, is
     do j = jl. ju. js
        do k = kl, ku, ks
           .....
        end do
     end do
  end do
!$omp end parallel do
```

# The sections directive - Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



# Overlap I/O and Processing/1

| Time | Input Thread | Processing Thread(s) | Output Thread |
|---|---|---|---|
| ↓ | 0 | | |
| | 1 | 0 | |
| | 2 | 1 | 0 |
| | 3 | 2 | 1 |
| | 4 | 3 | 2 |
| | 5 | 4 | 3 |
| | | 5 | 4 |
| | | | 5 |

# Overlap I/O and Processing/2

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) read_input(i);
         (void) signal_read(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_read(i);
         (void) process_data(i);
         (void) signal_processed(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_processed(i);
         (void) write_output(i);
     }
   }
} /*-- End of parallel sections --*/
```

Input Thread

Processing Thread(s)

Output Thread

# Single processor region/1

*This construct is ideally suited for I/O or initializations*

*Original Code*

```
.....
"read a[0..N-1]";
.....
```

*"declare A to be be shared"*

```
#pragma omp parallel
{
   .....
```

*one volunteer requested*

```
   "read a[0..N-1]";
```

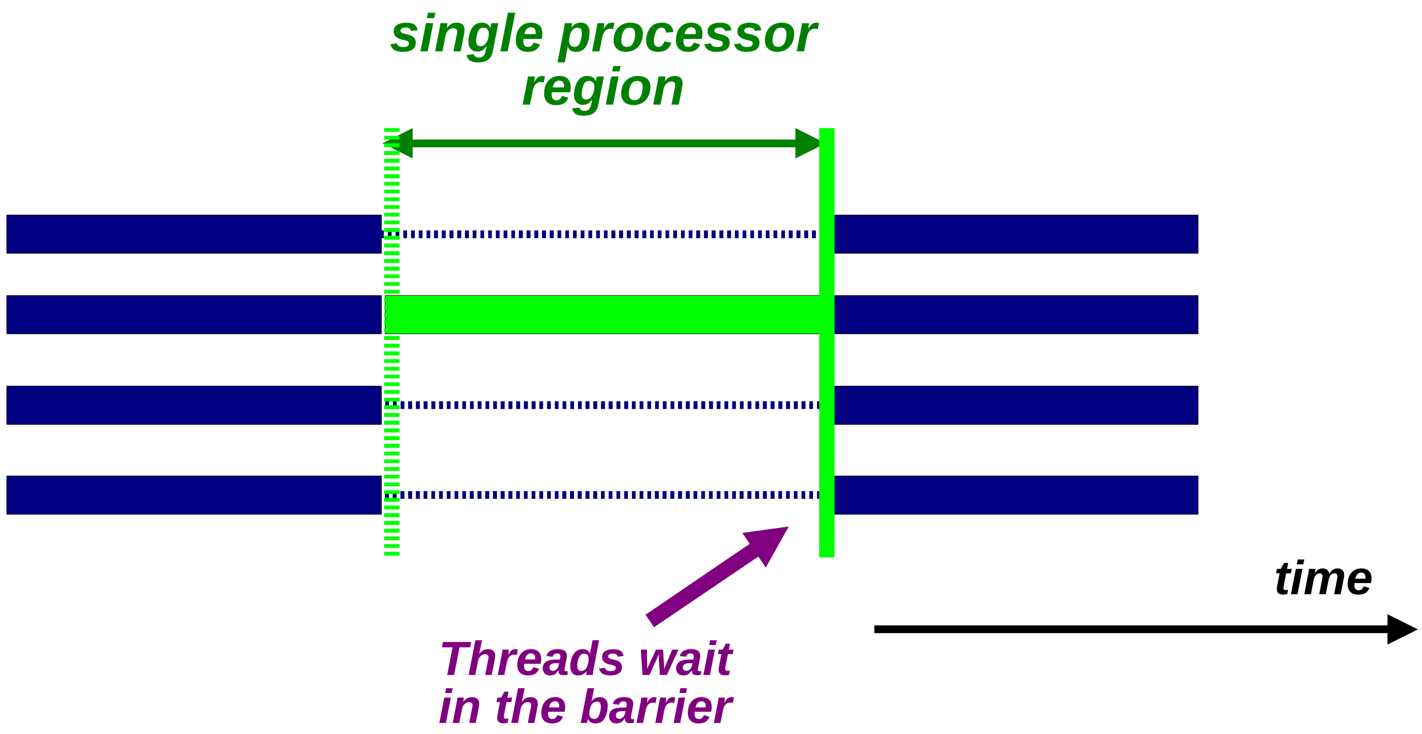
*thanks, we're done*

```
   .....
}
```

*Parallel Version*

May have to insert a barrier here

# Single processor region/2



# The Single Directive

*Only one thread in the team executes the code enclosed*

```
#pragma omp single [private][firstprivate] \
                     [copyprivate][nowait]
{
        <code-block>
}
```

```
!$omp single [private][firstprivate]
   <code-block>
!$omp end single [copyprivate][nowait]
```

# Combined work-sharing constructs

```
#pragma omp parallel
#pragma omp for
    for (...)
```

```
#pragma omp parallel for
for (....)
```

*Single PARALLEL loop*

```
!$omp parallel
!$omp do
      ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
      ...
!$omp end parallel do
```

*Single WORKSHARE loop*

```
!$omp parallel
!$omp workshare
      ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
      ...
!$omp end parallel workshare
```

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

*Single PARALLEL sections*

```
!$omp parallel
!$omp sections
      ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
      ...
!$omp end parallel sections
```

# Orphaning

```
           :
#pragma omp parallel
{
           :
      (void) dowork();
           :
}
           :
```

```
void dowork()
{
            :
  #pragma omp for
   for (int i=0;i<n;i++)
   {
            :
   }
            :
}
```

orphaned work-sharing directive

- *The OpenMP specif cation does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned*
- *That is, they can appear outside the lexical extent of a parallel region*

# More on orphaning

```
(void) dowork(); !- Sequential FOR

#pragma omp parallel
{
   (void) dowork(); !- Parallel FOR
}
```

```
void dowork()
{
#pragma omp for
   for (i=0;....)
   {
      :
   }
}
```

- *When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored*

# Parallelizing bulky loops

```
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

International Workshop IWOMP on OpenMP

# Step 1: “Outlining”

```
for (i=0; i<n; i++) /* Parallel loop */
{
     (void) FuncPar(i,m,c,...)
}
```

*Still a sequential program*

*Should behave identically*

*Easy to test for correctness*

*But, parallel by design*

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
          ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
          ......
}
```

# Step 2: Parallelize

```
#pragma omp parallel for private(i) shared(m,c,..)
  for (i=0; i<n; i++) /* Parallel loop */
  {
      (void) FuncPar(i,m,c,...)
  } /*-- End of parallel for --*/
```

*Minimal scoping required*

*Less error prone*

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

# OpenMP Runtime Routines

# OpenMP Runtime Functions/1

| Name | Functionality |
| --- | --- |
| omp_set_num_threads | Set number of threads |
| omp_get_num_threads | Number of threads in team |
| omp_get_max_threads | Max num of threads for parallel region |
| omp_get_thread_num | Get thread ID |
| omp_get_num_procs | Maximum number of processors |
| omp_in_parallel | Check whether in parallel region |
| omp_set_dynamic | Activate dynamic thread adjustment (but implementation is free to ignore this) |
| omp_get_dynamic | Check for dynamic thread adjustment |
| omp_set_nested | Activate nested parallelism (but implementation is free to ignore this) |
| omp_get_nested | Check for nested parallelism |
| omp_get_wtime | Returns wall clock time |
| omp_get_wtick | Number of seconds between clock ticks |

**C/C++ : Need to include file <omp.h>**
**Fortran : Add "use omp_lib" or include file "omp_lib.h"**

# OpenMP Runtime Functions/2

| *Name* | *Functionality* |
|---|---|
| *omp_set_schedule* | *Set schedule (if "runtime" is used)* |
| *omp_get_schedule* | *Returns the schedule in use* |
| *omp_get_thread_limit* | *Max number of threads for program* |
| *omp_set_max_active_levels* | *Set number of active parallel regions* |
| *omp_get_max_active_levels* | *Number of active parallel regions* |
| *omp_get_level* | *Number of nested parallel regions* |
| *omp_get_active_level* | *Number of nested active par. regions* |
| *omp_get_ancestor_thread_num* | *Thread id of ancestor thread* |
| *omp_get_team_size (level)* | *Size of the thread team at this level* |

**C/C++ : Need to include file <omp.h>**
**Fortran : Add "use omp_lib" or include file "omp_lib.h"**

# OpenMP locking routines

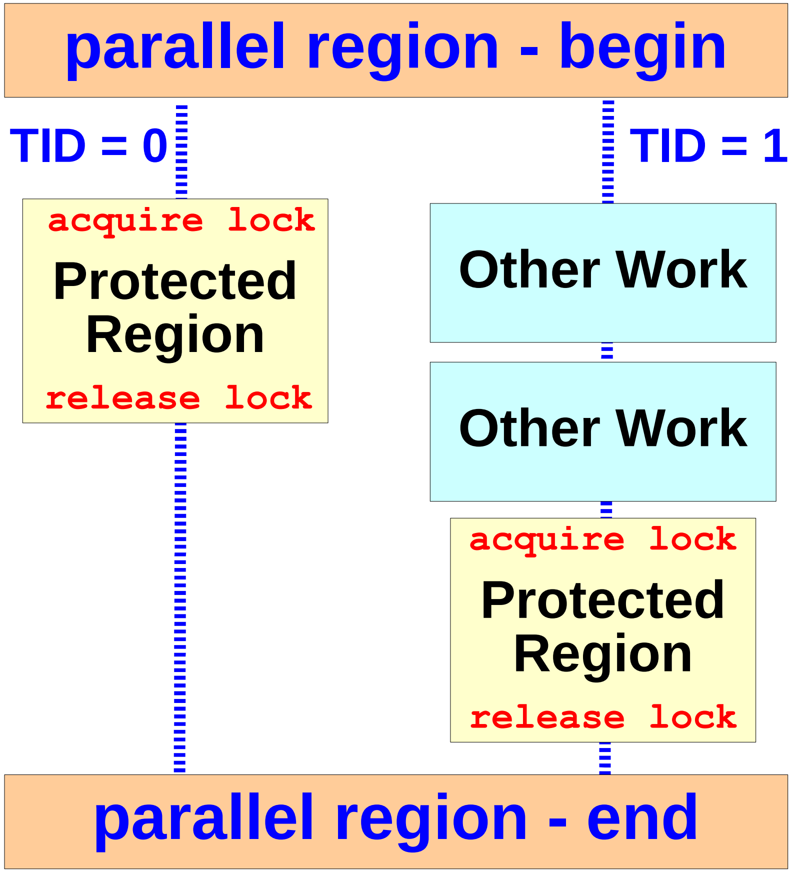
- ***Locks provide greater flexibility over critical sections and atomic updates:***
  - ***Possible to implement asynchronous behavior***
  - ***Not block structured***
- ***The so-called lock variable, is a special variable:***
  - ***Fortran: type INTEGER and of a KIND large enough to hold an address***
  - ***C/C++: type omp_lock_t and omp_nest_lock_t for nested locks***
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, <u>and behavior is undefined</u>, in case a lock variable is used without the appropriate initialization***

# Nested locking

- *Simple locks: may not be locked if already in a locked state*
- *Nestable locks: may be locked multiple times by the same thread before being unlocked*
- *In the remainder, we discuss simple locks only*
- *The interface for functions dealing with nested locks is similar (but using nestable lock variables):*

| Simple locks | Nestable locks |
| --- | --- |
| `omp_init_lock` | `omp_init_nest_lock` |
| `omp_destroy_lock` | `omp_destroy_nest_lock` |
| `omp_set_lock` | `omp_set_nest_lock` |
| `omp_unset_lock` | `omp_unset_nest_lock` |
| `omp_test_lock` | `omp_test_nest_lock` |

# OpenMP locking example




- *The protected region contains the update of a shared variable*
- *One thread acquires the lock and performs the update*
- *Meanwhile, the other thread performs some other work*
- *When the lock is released again, the other thread performs the update*

# Locking Example - The Code

```
      Program Locks
            ....
      Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else()
       End Do

       Call Do_Work()

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

# Example output for 2 threads

```
 TID:  1 at 09:07:27 => entered parallel region
 TID:  1 at 09:07:27 => done with WAIT loop and has the lock
 TID:  1 at 09:07:27 => ready to do the parallel work
 TID:  1 at 09:07:27 => this will take about 18 seconds
 TID:  0 at 09:07:27 => entered parallel region
 TID:  0 at 09:07:27 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:32 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:37 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:42 =>   WAIT for lock - will do something else for  5 seconds
 TID:  1 at 09:07:45 => done with my work
 TID:  1 at 09:07:45 => done with work loop - released the lock
 TID:  1 at 09:07:45 => ready to leave the parallel region
 TID:  0 at 09:07:47 => done with WAIT loop and has the lock
 TID:  0 at 09:07:47 => ready to do the parallel work
 TID:  0 at 09:07:47 => this will take about 18 seconds
 TID:  0 at 09:08:05 => done with my work
 TID:  0 at 09:08:05 => done with work loop - released the lock
 TID:  0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```

**Used to check the answer**

***Note: program has been instrumented to get this information***

# *OpenMP Environment Variables*

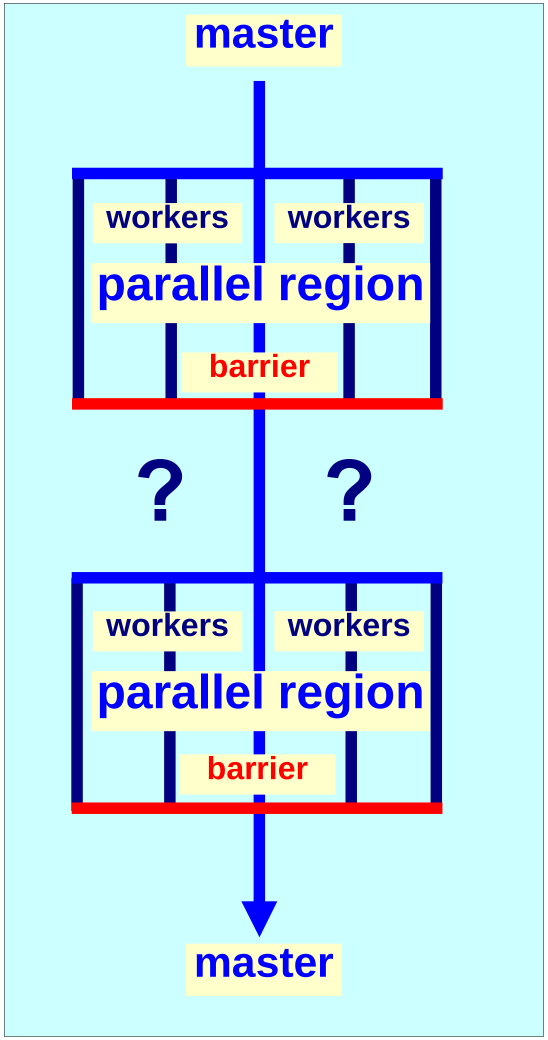
# OpenMP Environment Variables

| OpenMP environment variable | Default for Oracle Solaris Studio |
|---|---|
| OMP_NUM_THREADS n | 1 |
| OMP_SCHEDULE “schedule,[chunk]” | static, “N/P” |
| OMP_DYNAMIC { TRUE \| FALSE } | TRUE |
| OMP_NESTED { TRUE \| FALSE } | FALSE |
| OMP_STACKSIZE size [B\|K\|M\|G] | 4 MB (32 bit) / 8 MB (64-bit) |
| OMP_WAIT_POLICY [ACTIVE \| PASSIVE] | PASSIVE |
| OMP_MAX_ACTIVE_LEVELS | 4 |
| OMP_THREAD_LIMIT | 1024 |

*Note:*

*The names are in uppercase, the values are case insensitive*

# Implementing the Fork-Join Model

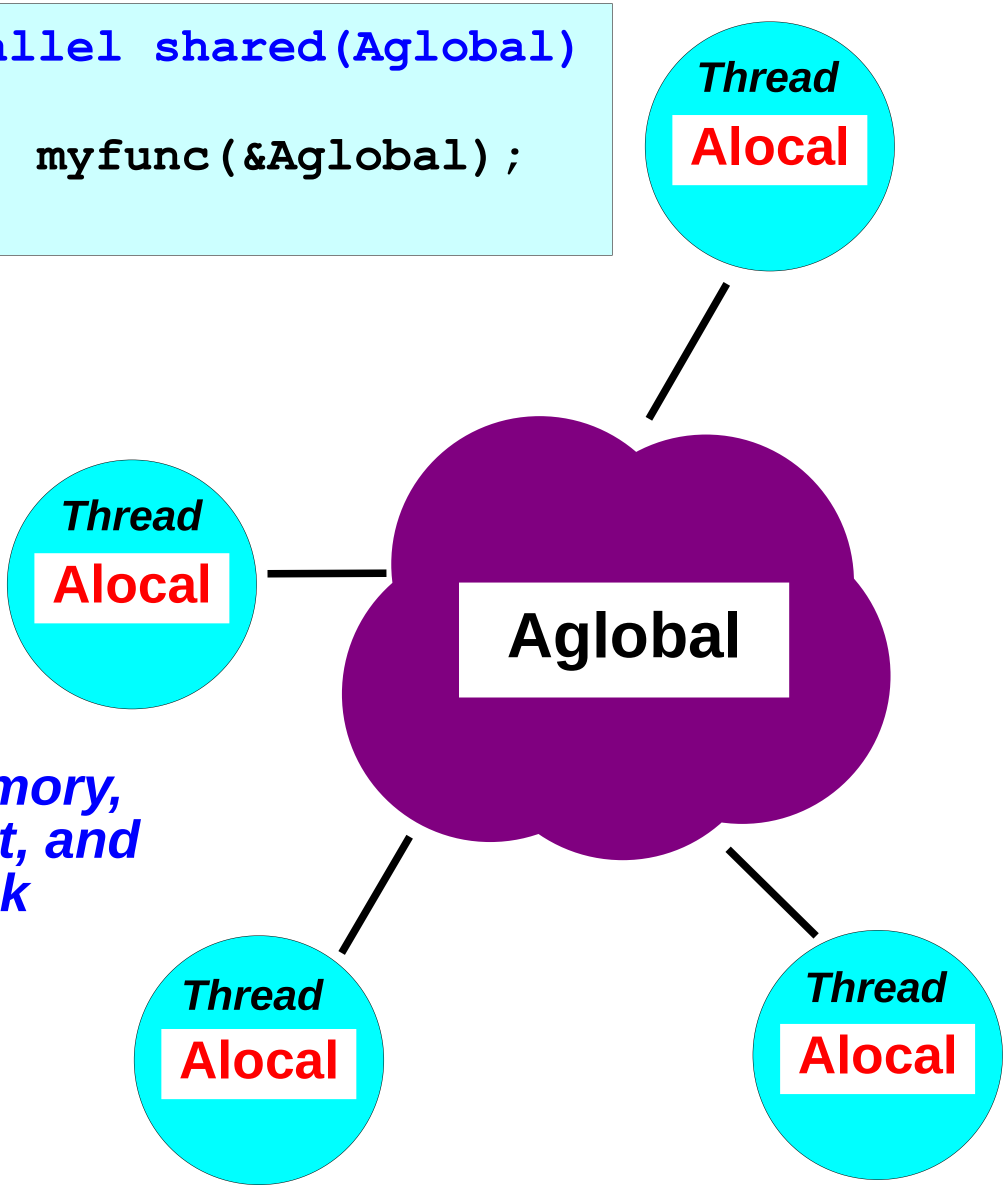



***Use OMP_WAIT_POLICY to control behaviour of idle threads***

# About the stack

```
#omp parallel shared(Aglobal)
{
    (void) myfunc(&Aglobal);
}
```

```
void myfunc(float *Aglobal)
{
    int Alocal;

    .......
}
```




***Variable* Alocal *is in private memory, managed by the thread owning it, and stored on the so-called stack***

# Setting the size of the stack

```
OMP_STACKSIZE n [B,K,M,G]
```

*Set thread stack size in n Byte, KB, MB, or GB*
*Default is KByte*

- ☞ *Each thread has its own private stack space*
- ☞ *If a thread runs out of this stack space, the behavior is undef ned*
- ☞ *Note there are two stack sizes involved:*
  - ✔ ***Master Thread - Use the appropriate OS command (e.g. in Unix "limit/ulimit") to its stack size***
  - ✔ ***Worker Threads - Use the OMP_STACKSIZE environment variable to increase the stack size for each of the worker threads***
- ☞ *The default value for OMP_STACKSIZE is implementation dependent*

# Example OMP_STACKSIZE

```
#define N 2000000

void myFunc(int TID, double *check);

void main()
{
   double check, a[N];

#pragma omp parallel private(chec
  {
    myFunc(&check);

  } /*-- End of parallel region

}
```

*Main requires about 16 MByte stack space to run*

```
#define MYSTACK 1000000

void myFunc(double *check)
{
   double mystack[MYSTACK];
   int    i;

   for (i=0; i<MYSTACK; i++)
      mystack[i] = TID + 1;

   *check = mystack[MYSTACK-1];
           ..........
}
```

*Function requires about ~8 MByte stack space to run*

International Workshop IWOMP on OpenMP

# Run-time Behaviour

```
% setenv OMP_NUM_THREADS 1
% limit stack 10k
% ./stack.exe
Segmentation Fault (core dumped)
% limit stack 16m
% ./stack.exe
Thread 0 has initialized local data
```

*Not enough stack space for master thread*

*Now runs fine on 1 thread*

```
% setenv OMP_NUM_THREADS 2
% ./stack.exe
Segmentation Fault (core dumped)
% setenv OMP_STACKSIZE 8192
% setenv OMP_NUM_THREADS 1
% ./stack.exe
Thread 0 has initialized local data
% setenv OMP_NUM_THREADS 2
% ./stack.exe
Thread 0 has initialized local data
Thread 1 has initialized local data
% setenv OMP_NUM_THREADS 4
% ./stack.exe
Thread 0 has initialized local data
Thread 2 has initialized local data
Thread 3 has initialized local data
Thread 1 has initialized local data
```

*But crashes on 2 ....*

*Increase thread stacksize and all is well again*

# Using OpenMP

# Using OpenMP

- *We have already seen many features of OpenMP*
- *We will now cover*
  - *Additional language constructs*
  - *Features that may be useful or needed when running an OpenMP application*
- *The tasking concept is covered in separate section*

# About storage association

- ***<u>Private variables are undefined on entry and exit of the parallel region</u>***
- ***A private variable within a parallel region has <u>no storage association</u> with the same variable outside of the region***
- ***Use the first/last private clause to override this behavior***
- ***We illustrate these concepts with an example***

# Example private variables

```
main()
{
  A = 10;

 #pragma omp parallel
 {
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;          /*-- A undefined, unless declared
                               firstprivate --*/
      ....
  }

  C = B;                  /*-- B undefined, unless declared
                               lastprivate --*/

 } /*-- End of OpenMP parallel region --*/
}
```

*Disclaimer: This code fragment is not very meaningful and only serves to demonstrate the clauses*

# The first/last private clauses

**firstprivate (list)**

- ✔ ***All variables in the list are initialized with the value the original object had before entering the parallel construct***

**lastprivate (list)**

- ✔ ***The thread that executes the <u>sequentially last</u> iteration or section updates the value of the objects in the list***

# The default clause

default (none | shared | private | threadprivate ) *Fortran*

default ( none | shared ) *C/C++*

**none**

- ✔ *No implicit defaults; have to scope all variables explicitly*

**shared**

- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

**private**

- ✔ *All variables are private to the thread*
- ✔ *Includes common block data, unless THREADPRIVATE*

**ﬁ rstprivate**

- ✔ *All variables are private to the thread; pre-initialized*

# The reduction clause - Example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

*Variable SUM is a shared variable*

- *Care needs to be taken when updating shared variable SUM*
- *With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided*

# The reduction clause

**reduction ( [operator | intrinsic] ) : list )** *Fortran*

**reduction ( operator : list )** *C/C++*

- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is def ned as:*

*Check the docs for details*

| Fortran | C/C++ |
|---|---|
| `x = x operator expr` | `x = x operator expr` |
| `x = expr operator x` | `x = expr operator x` |
| `x = intrinsic (x, expr_list)` | `x++, ++x, x--, --x` |
| `x = intrinsic (expr_list, x)` | `x <binop> = expr` |

- ✔ *Note that the value of a reduction variable is undef ned from the moment the f rst thread reaches the clause till the operation has completed*
- ✔ *The reduction can be hidden in a function call*

# Fortran - Allocatable Arrays

- ***Fortran allocatable arrays whose status is "currently allocated" are allowed to be specified as private, lastprivate, firstprivate, reduction, or copyprivate***

```
integer, allocatable,dimension (:) :: A
integer i

allocate (A(n))

!$omp parallel private (A)
      do i = 1, n
         A(i) = i
      end do
      ...
!$omp end parallel
```

# The schedule clause/1

**schedule ( static | dynamic | guided | auto [, chunk] )**
**schedule (runtime)**

**static [, chunk]**

- ✔ ***Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion***
- ✔ ***In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads***
  - ***Details are implementation defined***
- ✔ ***Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region***

# The schedule clause/2

## *Example static schedule*

### *Loop of length 16, 4 threads:*

| Thread | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| *no chunk** | 1-4 | 5-8 | 9-12 | 13-16 |
| *chunk = 2* | 1-2<br>9-10 | 3-4<br>11-12 | 5-6<br>13-14 | 7-8<br>15-16 |

****) The precise distribution is implementation def ned***

# The schedule clause/3

**dynamic [, chunk]**

- ✔ *Fixed portions of work; size is controlled by the value of chunk*
- ✔ *When a thread finishes, it starts on the next portion of work*

**guided [, chunk]**

- ✔ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

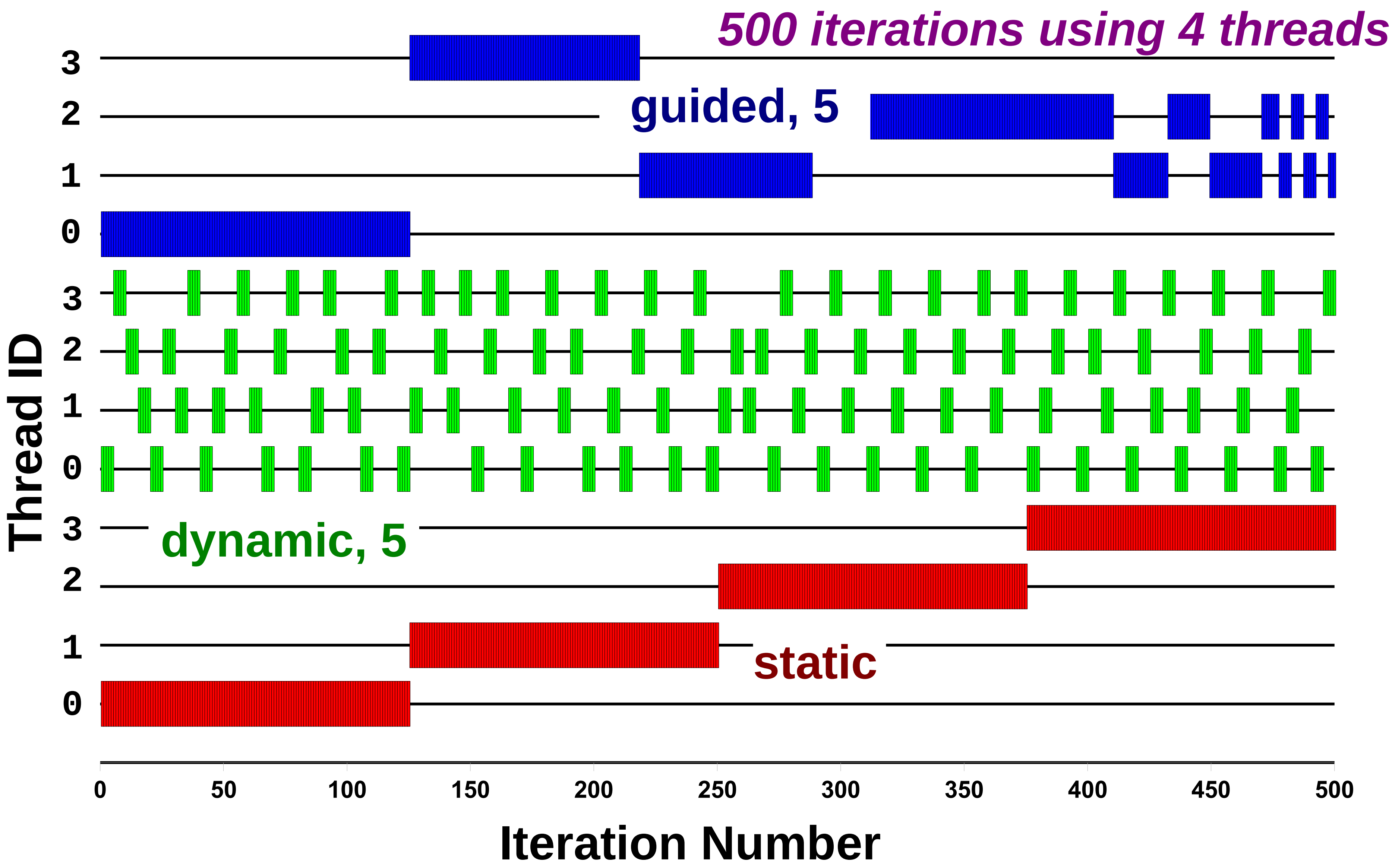
**auto**

- ✔ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

**runtime**

- ✔ *Iteration scheduling scheme is set at runtime through environment variable **OMP_SCHEDULE***

# The Experiment



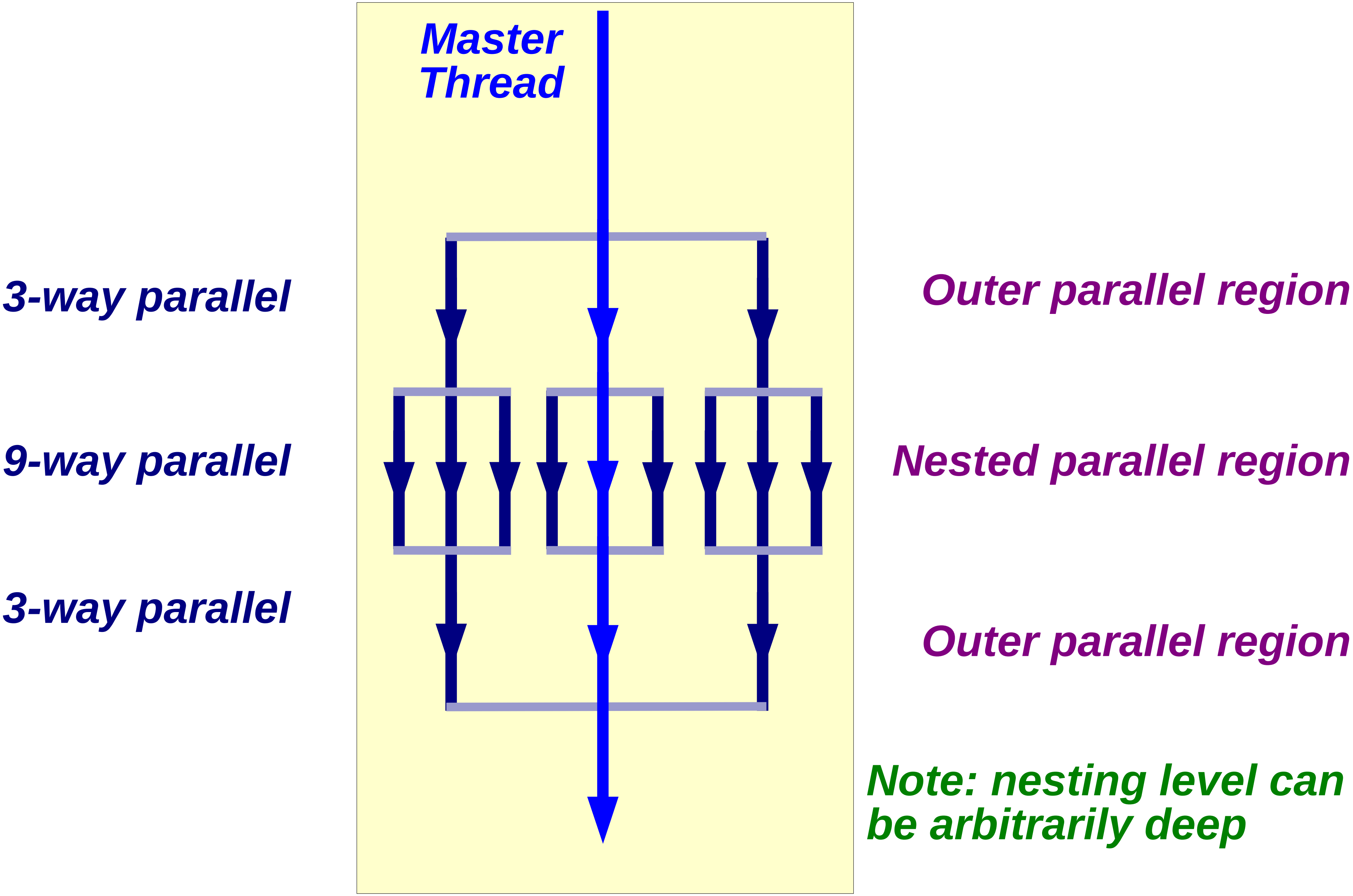
# Schedule Kinds Functions

- *Makes* `schedule(runtime)` *more general*
- *Can set/get schedule it with library routines:*

```
omp_set_schedule()
omp_get_schedule()
```

- *Also allows implementations to add their own schedule kinds*

# Nested Parallelism




***Note: nesting level can be arbitrarily deep***

# Nested Parallelism Support/1

- *Environment variable and runtime routines to set/get the maximum number of nested active parallel regions*

  ***OMP_MAX_ACTIVE_LEVELS***

  ***omp_set_max_active_levels()***

  ***omp_get_max_active_levels()***

- *Environment variable and runtime routine to set/get the maximum number of OpenMP threads available to the program*

  ***OMP_THREAD_LIMIT***

  ***omp_get_thread_limit()***

# Nested Parallelism Support/2

- *Per-task internal control variables*
  - *Allow, for example, calling* ***omp_set_num_threads()*** *inside a parallel region to control the team size for next level of parallelism*
- *Library routines to determine*
  - *Depth of nesting*

    ```
    omp_get_level()
    omp_get_active_level()
    ```

  - *IDs of parent/grandparent etc. threads*

    ```
    omp_get_ancestor_thread_num(level)
    ```

  - *Team sizes of parent/grandparent etc. teams*

    ```
    omp_get_team_size(level)
    ```

# Additional Directives/1

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

```
#pragma omp atomic
```

```
!$omp atomic
```

# Additional Directives/2

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
       <code-block>
!$omp end ordered
```

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

# The Master Directive

***Only the <u>master thread</u> executes the code block:***

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

***There is no implied barrier on entry or exit !***

# Critical Region/1

*If sum is a shared variable, this loop can not run in parallel*

```
for (i=0; i < n; i++){
     .....
   sum += a[i];
     .....
}
```

*We can use a critical region for this:*

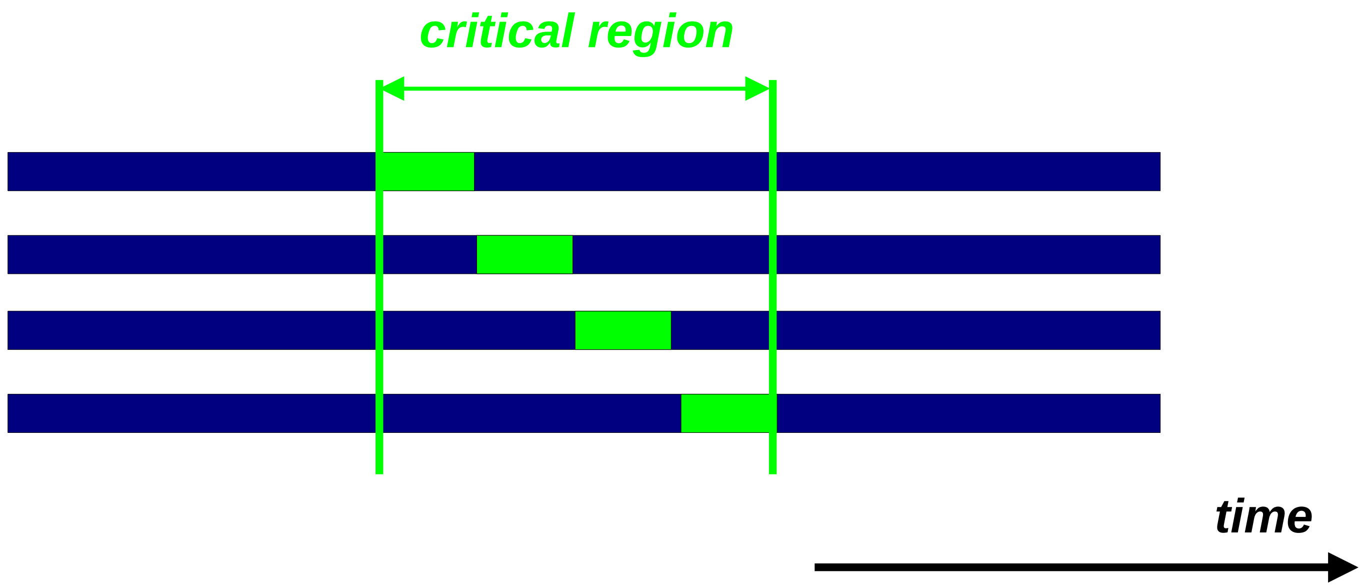
```
for (i=0; i < n; i++){
     .....
   sum += a[i];
     .....
}
```

*one at a time can proceed*

*next in line, please*

# Critical Region/2

- *Useful to avoid a race condition, or to perform I/O (but that still has random order)*
- *Be aware that there is a cost associated with a critical region*



# Critical and Atomic constructs

*Critical: All threads execute the code, but only one at a time:*

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

*There is no implied barrier on entry or exit !*

*Atomic: only the loads and store are atomic ....*

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

*This is a lightweight, special form of a critical section*

```
#pragma omp atomic
    a[indx[i]] += b[i];
```

International Workshop IWOMP on OpenMP

# More synchronization constructs

*The enclosed block of code is executed in the order in which iterations would be executed sequentially:*

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
        <code-block>
!$omp end ordered
```

*May introduce serialization (could be expensive)*

*Ensure that all threads in a team have a consistent view of certain objects in memory:*

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

*In the absence of a list, all visible variables are f ushed*

# Implied flush regions

- During a barrier region
- At exit from worksharing regions, unless a nowait is present
- At entry to and exit from parallel, critical, ordered and parallel worksharing regions
- During omp_set_lock and omp_unset_lock regions
- During omp_test_lock, omp_set_nest_lock, omp_unset _nest_lock and omp_test_nest_lock regions, if the region causes the lock to be set or unset
- Immediately before and after every task scheduling point
- At entry to and exit from atomic regions, where the list contains only the variable updated in the atomic construct
- A flush region is <u>not implied</u> at the following locations:
  - At entry to a worksharing region
  - At entry to or exit from a master region

# OpenMP and Global Data

# Global data - An example

```
      program global_data
             ....
      include "global.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
```

*f le global.h*

```
common /work/a(m,n),b(m)
```

```
subroutine suba(j)
  .....
include "global.h"
  .....
do i = 1, m
   b(i) = j
end do

do i = 1, m
    a(i,j) = func_call(b(i))
end do

return
end
```

*Data Race !*

# Global data - A Data Race!

**Thread 1**

```
call suba(1)

subroutine suba(j=1)
do i = 1, m
   b(i) = 1
end do
      ....
do i = 1, m
 a(i,1)=func_call(b(i))
end do
```

**Thread 2**

```
call suba(2)

subroutine suba(j=2)
do i = 1, m
   b(i) = 2
end do
      ....
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```

*Shared*

# Example - Solution

```
      program global_data
             ....
      include "global_ok.h"
          ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
        ......
```

*f le global_ok.h*

```
integer, parameter:: nthreads=4
common /work/a(m,n)
common /tprivate/b(m,nthreads)
```

```
subroutine suba(j)
  .....
include "global_ok.h"
  .....

TID = omp_get_thread_num()+1
do i = 1, m
   b(i,TID) = j
end do

do i = 1, m
   a(i,j)=func_call(b(i,TID))
end do

return
end
```

- ☞ *By expanding array B, we can give each thread unique access to it's storage area*
- ☞ *Note that this can also be done using dynamic memory (allocatable, malloc, ....)*

# About global data

- ***Global data is shared and requires special care***
- ***A problem may arise in case multiple threads access the same memory section simultaneously:***
  - ***Read-only data is no problem***
  - ***Updates have to be checked for race conditions***
- ***It is your responsibility to deal with this situation***
- ***In general one can do the following:***
  - ***Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel***
  - ***Manually create thread private copies of the latter***
  - ***Use the thread ID to access these private copies***
- ***Alternative: Use OpenMP's threadprivate directive***

# The threadprivate directive

- ❑ ***OpenMP's threadprivate directive***

```
!$omp threadprivate (/cb/ [,/cb/] ...)
```

```
#pragma omp threadprivate (list)
```

- ❑ ***Thread private copies of the designated global variables and common blocks are created***
- ❑ ***Several restrictions and rules apply when doing this:***
  - *The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)*
    - ✔ *Sun implementation supports changing the number of threads*
  - *Initial data is undef ned, unless* **copyin** *is used*
  - *......*
- ❑ ***Check the documentation when using threadprivate !***

# Example - Solution 2

```
      program global_data
             ....
      include "global_ok2.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
        ......
      stop
      end
```

*f le global_ok2.h*

```
common /work/a(m,n)
common /tprivate/b(m)
!$omp threadprivate(/tprivate/)
```

```
subroutine suba(j)
  .....
include "global_ok2.h"
  .....

do i = 1, m
   b(i) = j
end do

do i = 1, m
   a(i,j) = func_call(b(i))
end do

return
end
```

- ***The compiler creates thread private copies of array B, to give each thread unique access to it's storage area***
- ***Note that the number of copies is automatically adjusted to the number of threads***

# The copyin clause

**copyin (list)**

- ✔ *Applies to THREADPRIVATE common blocks only*
- ✔ *At the start of the parallel region, data of the master thread is copied to the thread private copies*

*Example:*

```
      common /cblock/velocity
      common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel           &
!$omp default (private)  &
!$omp copyin ( /cblock/, zfield )
```

# C++ and Threadprivate

- *As of OpenMP 3.0, it was clarified where/how threadprivate objects are constructed and destructed*
- *Allow C++ static class members to be threadprivate*

```
class T {
  public:
  static int i;
  #pragma omp threadprivate(i)
  ...
};
```

# Tasking in OpenMP

# What Is A Task?

***A TASK***

*"A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct"*

*COMMENT: When a thread executes a task, it produces a task region*

***TASK REGION***

*"A region consisting of all code encountered during the execution of a task"*

*COMMENT: A parallel region consists of one or more implicit task regions*

***EXPLICIT TASK***

*"A task generated when a task construct is encountered during execution"*

# Tasking Directives

```
#pragma omp task
```

```
!$omp task
```

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

# "Hello World/1"

```
$ cc hello.c
$ ./a.out
Hello World
$
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

          printf("Hello ");
          printf("World ");

   printf("\n");
   return(0);
}
```

***What will this program print ?***

# "Hello World/2"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello ");
          printf("World ");

   } // End of parallel region

   printf("\n");
   return(0);
}
```

***What will this program print using 2 threads ?***

# "Hello World/3"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World Hello World
```

***Note that this program could also print "Hello Hello World World", although I have not observed it (yet)***

# "Hello World/4"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
          printf("Hello ");
          printf("World ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

***What will this program print using 2 threads ?***

# "Hello World/5"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
```

***But now only 1 thread executes .......***

# “Hello World/6”

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

```
int main(int argc, ch

   #pragma omp parallel
   {
     #pragma omp single
     {
        #pragma omp task
          {printf("Hello ");}
        #pragma omp task
          {printf("World ");}
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

***What will this program print using 2 threads ?***

# "Hello World/7"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
$ ./a.out
Hello World
$ ./a.out
World Hello
$
```

***Tasks can be executed in arbitrary order***

# “Hello World/8”

```
int main(int argc, ch

    #pragma omp parallel
    {
      #pragma omp single
      {
         #pragma omp task
           {printf("Hello ");}
         #pragma omp task
           {printf("World ");}
         printf(“\nThank You “);
      }
    } // End of parallel region

    printf("\n");
    return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

***What will this program print using 2 threads ?***

# “Hello World/9”

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

Thank You World Hello
$ ./a.out

Thank You Hello World
$ ./a.out

Thank You World Hello
$
```

***Tasks are executed at a task execution point***

# “Hello World/10”

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

```
int main(int argc, cha

   #pragma omp parallel
   {
     #pragma omp single
     {
        #pragma omp task
          {printf("Hello ");}
        #pragma omp task
          {printf("World ");}
        #pragma omp taskwait
        printf(“\nThank You “);
     }
   } // End of parallel region

   printf("\n");return
}
```

***What will this program print using 2 threads ?***

# “Hello World/11”

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
World Hello
Thank You
$ ./a.out
World Hello
Thank You
$ ./a.out
Hello World
Thank You
$
```

***Tasks are executed first now***

# Task Construct Syntax

***C/C++*:**

```
#pragma omp task [clause [[,]clause] ...]
   structured-block
```

***Fortran*:**

```
!$omp task[clause [[,]clause] ...]
   structured-block
!$omp end task
```

# Task Synchronization

- *Syntax:*
  - *C/C++:* `#pragma omp taskwait`
  - *Fortran:* `!$omp taskwait`
- *Current task suspends execution until all children tasks, generated within the current task up to this point, have completed execution*

# When are Tasks Complete?

- ***At implicit thread barrier***
- ***At explicit thread barrier***
  - *C/C++:* `#pragma omp barrier`
  - *Fortran:* `!$omp barrier`
- ***At task barrier***
  - *C/C++:* `#pragma omp taskwait`
  - *Fortran:* `!$omp taskwait`

# *Tasking Examples*

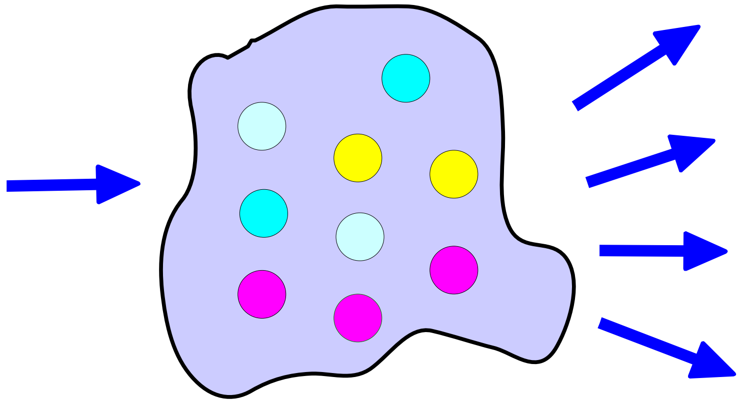
# Example - A Linked List

```
        ........
while(my_pointer) {

    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
        ........
```

***Hard to do before OpenMP 3.0:***
***First count number of iterations, then convert while loop to for loop***

# The Tasking Example

Encountering thread adds task(s) to pool



Threads execute tasks in the pool

***Developer specifies tasks in application***
***Run-time system executes tasks***

# Example - A Linked List With Tasking

OpenMP Task is specified here (executed in parallel)

```
my_pointer = listhead;

#pragma omp parallel
{
   #pragma omp single nowait
   {
      while(my_pointer) {
        #pragma omp task firstprivate(my_pointer)
        {
            (void) do_independent_work (my_pointer);
        }
        my_pointer = my_pointer->next ;
      }
   } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

# Example – Fibonacci Numbers

***The Fibonacci Numbers are defined as follows:***

```
F(0) = 1
F(1) = 1

F(n) = F(n-1) + F(n-2) (n=2,3,4,.....)
```

***Sequence:***

***1, 1, 2, 3, 5, 8, 13, 21, 34, ....***

# Recursive Algorithm*

```
long comp_fib_numbers(int n){
   // Basic algorithm: f(n) = f(n-1) + f(n-2)
   long fnm1, fnm2, fn;
   if ( n == 0 || n == 1 ) return(n);

   fnm1 = comp_fib_numbers(n-1);

   fnm2 = comp_fib_numbers(n-2);

   fn   = fnm1 + fnm2;
   return(fn);
}
```

***) Not very efficient, used for demo purposes only**

# Parallel Recursive Algorithm

```
long comp_fib_numbers(int n){
   // Basic algorithm: f(n) = f(n-1) + f(n-2)
   long fnm1, fnm2, fn;
   if ( n == 0 || n == 1 ) return(n);
#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}
#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}
#pragma omp taskwait
   fn   = fnm1 + fnm2;
   return(fn);
}
```

# Driver Program

```
#pragma omp parallel shared(nthreads)
{
   #pragma omp single nowait
   {

      result = comp_fib_numbers(n);

   } // End of single
} // End of parallel region
```

# Parallel Recursive Algorithm - V2

```
long comp_fib_numbers(int n){

   // Basic algorithm: f(n) = f(n-1) + f(n-2)

   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 ) return(n);
   if ( n < 20 ) return(comp_fib_numbers(n-1) +
                        comp_fib_numbers(n-2));

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

# Performance Example*

```
$ export OMP_NUM_THREADS=1
$ ./fibonacci-omp.exe 40
Parallel   result for n = 40: 102334155 (1 threads
                              needed 5.63 seconds)
$ export OMP_NUM_THREADS=2
$ ./fibonacci-omp.exe 40
Parallel   result for n = 40: 102334155 (2 threads
                              needed 3.03 seconds)
$
```

***) MacBook Pro Core 2 Duo***

# Oracle Solaris Studio Support for OpenMP Development

# OpenMP Compiler Options

| Option | Description |
| --- | --- |
| -xopenmp | Equivalent to -xopenmp=parallel |
| -xopenmp=parallel | Enables recognition of OpenMP pragmas<br>Requires at least optimization level -xO3 |
| -xopenmp=noopt | Enables recognition of OpenMP pragmas<br>The program is parallelized accordingly, but no optimization is done * |
| -xopenmp=none | Disables recognition of OpenMP pragmas (default) |

**) The compiler does not raise the optimization level if it is lower than -xO3*

# Related Compiler Options

| Option | Description |
|---|---|
| -xloopinfo | Display parallelization messages on screen |
| -stackvar | Allocate local data on the stack (Fortran only)<br>Use this when calling functions in parallel<br>Included with -xopenmp=parallel \| noopt |
| -vpara/-xvpara | Reports OpenMP scoping errors in case of incorrect parallelization (Fortran and C compiler only)<br>Also reports OpenMP scoping errors and race conditions statically detected by the compiler |
| -XlistMP | Reports warnings about possible errors in OpenMP parallelization (Fortran only) |

# Support for Threadprivate

- ❑ *It can be tedious to implement THREADPRIVATE*
- ❑ *The Oracle Solaris Studio Fortran compiler supports the **-xcommonchk** option to report upon inconsistent usage of threadprivate*
  - *Common block declared THREADPRIVATE in one program unit, but not in another*
  - *Does not check for consistency on the size of the common block*
- ❑ *Syntax:* `-xcommonchk [={yes|no}]`
- ❑ *Run-time checks are inserted, causing performance to degrade*
- ❑ *This is therefore a debugging option*

# Run-time warnings

| | |
|---|---|
| `SUNW_MP_WARN   TRUE | FALSE` | *Control printing of warnings* |

- ☞ *The OpenMP run-time library does not print warning messages by default*
- ☞ *Strongly recommended to set this environment variable to TRUE to activate the warnings*
- ☞ *This helps to diagnose run-time problems*
  - *Also reports (some) non-conforming program errors*
- ☞ *Note there is a slight performance penalty associated with setting this environment variable to TRUE*
  - *Cost depends on the operation - Explicit locking is more expensive for example*

# Example SUNW_MP_WARN/1

## *Using more threads than processors:*

```
# SUNW_MP_WARN=TRUE; export SUNW_MP_WARN
# OMP_NUM_THREADS=3; export OMP_NUM_THREADS
# ./omp.exe
WARNING (libmtsk): Dynamic adjustment of threads is enabled. The
number of threads is adjusted to 2.
Thread ID 0 updates i = 0
Thread ID 0 updates i = 1
Thread ID 0 updates i = 2
Thread ID 1 updates i = 3
Thread ID 1 updates i = 4
Thread ID 1 updates i = 5
```

***Now we get 3 threads***

```
# OMP_DYNAMIC=FALSE; export OMP_DYNAMIC
# ./omp.exe
Thread ID 0 updates i = 0
Thread ID 0 updates i = 1
Thread ID 1 updates i = 2
Thread ID 1 updates i = 3
Thread ID 2 updates i = 4
Thread ID 2 updates i = 5
```

# Example SUNW_MP_WARN/2

```
void foo()
{
  #pragma omp barrier
     whatever();
}

void bar(int n)
{
  printf("In bar: n = %d\n",n);
  #pragma omp parallel for
  for (int i=0; i<n; i++)
     foo();
}

void whatever()
{
  int TID = omp_get_thread_num();
  printf("Thread %d does nothing\n",TID);
}
```

# Example SUNW_MP_WARN/3

```
% cc -fast -xopenmp -xloopinfo -xvpara main.c
"main.c", line 30: PARALLELIZED, user pragma used
% setenv OMP_NUM_THREADS 4
% setenv SUNW_MP_WARN TRUE
% ./a.out
In bar: n = 5
WARNING (libmtsk): at main.c:22. Barrier is not permitted in
dynamic extent of for / DO.
Thread 0 does do nothing
Thread 3 does do nothing
Thread 2 does do nothing
Thread 1 does do nothing
WARNING (libmtsk): Threads at barrier from different
directives.
    Thread at barrier from main.c:22.
    Thread at barrier from main.c:29.
    Possible Reasons:
    Worksharing constructs not encountered by all threads in
    the team in the same order.
    Incorrect placement of barrier directives.
Thread 0 does do nothing
```

***Application hangs***

# Thread Aff nity (experimental)

```
SUNW_MP_THR_AFFINITY  TRUE | FALSE
```

***Improve thread aff nity***

***If set to TRUE, the master thread no longer returns worker threads to the pool***

# Processor binding

*Control binding of threads to "processors"*

```
SUNW_MP_PROCBIND  TRUE | FALSE
SUNW_MP_PROCBIND  Logical ID, or Range of logical IDs,
                  or list of logical IDs (separated by
                  spaces)
```

- ☞ ***Processor binding, when used along with static scheduling, benef ts applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region is in the local cache from a previous invocation of a parallel region***
- ☞ ***One can use the psrinfo and prtdiag (in /usr/sbin) commands to find out how processors are configured***
- ☞ ***Note that the binding is to the logical processor ID, not the physical ID (order is dictated by output of psrinfo)***
- ☞ ***In case of syntax error, an error message is emitted and execution of the program is terminated.***

# Default Stack Traceback

```
% ./stack.exe
Segmentation Fault (core dumped)
% pstack core
core 'core' of 10043:    ./stack.exe
-----------------  lwp# 2 / thread# 2  --------------------
 00010850 myFunc   (1, fe3ffda0, 0, 1, 0, 0) + 10
 0001082c _$p1A19.main (0, fe793380, 80, 10820, feb68260, 0) + c
 feb6834c run_job_invoke_mfunc_once (fe793380, 0, ffbff9a8, 1, 0, 0) + ac
 feb686b4 run_my_job (fe793380, 0, ffbff9a8, 2, 1, 27395000) + 20
 feb736a4 slave_startup_function (feb97290, fe7933d0, fe7933a8, 1, 2,
          feb97284) + 7dc
 feb457b4 _lwp_start (0, 0, 0, 0, 0, 0)
-----------------  lwp# 1 / thread# 1  --------------------
 000108ac myFunc   (f4238, ffbff698, 0, ffbff698, 1438, ff4685f0) + 6c
 0001082c _$p1A19.main (0, fe782100, 80, 10820, feb68260, 0) + c
 feb6834c run_job_invoke_mfunc_once (fe782100, 0, ffbff9a8, 1, ffbff768,
          ffbff879) + ac
 feb67914 __mt_MasterFunction_rtc_ (107a0, fe782180, 0, 13, fe782334, 0) +
          51c
 0001080c main     (1, 13, 702, 107a0, 10400, 10820) + 4c
 00010788 _start   (0, 0, 0, 0, 0, 0) + 108
```

*pstack is a very useful Solaris command !*

# Compiler option: -xcheck=stkovf

```
% cc -o stack_stkovf.exe -fast -g -xopenmp -xcheck=stkovf *.c
% ./stack_stkovf.exe
Segmentation Fault (core dumped)
% pstack core
core 'core' of 10077:   ./stack_stkovf.exe

-----------------  lwp# 2 / thread# 2  --------------------
 feb45bb4 _stack_grow (1, fe3ffda0, 0, 1, 0, 0) + 48
 00010890 _$p1A19.main (0, fe793380, 80, 10880, feb68260, 0) + 10
 feb6834c run_job_invoke_mfunc_once (fe793380, 0, ffbff98[illegible], 0) + ac
 feb686b4 run_my_job (fe793380, 0, ffbff988, 2, 1, 2739[illegible]
 feb736a4 slave_startup_function (feb97290, fe7933d0, fe[illegible]0,
          feb97284) + 7dc
 feb457b4 _lwp_start (0, 0, 0, 0, 0, 0)
-----------------  lwp# 1 / thread# 1  --------------------
 00010904 myFunc   (f4238, ffbff678, 0, ffbff678, 1340, ff467e1[illegible]
 00010890 _$p1A19.main (0, fe782100, 80, 10880, feb68260, 0) + 10
 feb6834c run_job_invoke_mfunc_once (fe782100, 0, ffbff988, 1, ffbf[illegible]
          ffbff859) + ac
 feb67914 __mt_MasterFunction_rtc_ (10800, fe782180, 0, 13, fe782334, 0) +
          51c
 00010870 main     (1, 13, 702, 10800, 10800, 10880) + 50
 000107e8 _start   (0, 0, 0, 0, 0, 0) + 108
```

Currently supported only on SPARC

# The behavior of idle threads

## *Environment variable to control the behavior:*

```
SUNW_MP_THR_IDLE
     [ spin | sleep [('n's) , ('n'ms) , ('n'mc)] ]
```

- ***Default is to have idle threads go to sleep after a spinning for a short while***
- ***Spin: threads keep the CPU busy (but don't do useful work)***
- ***Sleep: threads are put to sleep; awakened when new work arrives***
- ***Sleep ('time'): spin for 'n' seconds ( or milli/micro seconds), then go into sleep mode***
  - ***Examples:*** ***setenv SUNW_MP_THR_IDLE "sleep(5 ms)"***
    ***setenv SUNW_MP_THR_IDLE spin***

# *Autoscoping*

# Autoscoping example

*Autoscoping is a unique feature available in the Oracle Solaris Studio compilers only*

```
!$OMP PARALLEL DEFAULT (__AUTO)

!$OMP SINGLE
      T = N*N
!$OMP END SINGLE

!$OMP DO
      DO I = 1, N
         A(I) = T + I
      END DO
!$OMP END DO

!$OMP END PARALLEL
```

# Autoscoping results

```
Shared variables in OpenMP construct below: a, i, t, n
Variables autoscoped as SHARED in OpenMP construct below: i, t, n, a

 10. !$OMP PARALLEL DEFAULT (__AUTO)
 11.
 12. !$OMP SINGLE
 13.       T = N*N
 14. !$OMP END SINGLE
 15.

Private variables in OpenMP construct below: i
 16. !$OMP DO

Loop below parallelized by explicit user directive
 17.       DO I = 1, N
     <Function: _$d1A16.auto_>
 18.          A(I) = T + I
 19.       END DO
 20. !$OMP END DO
 21.
 22. !$OMP END PARALLEL
```

Variable 'i' re-scoped

# Example Autoscoping in C

```
$ suncc -c -fast -xrestrict -g -xopenmp -xloopinfo auto.c
"auto.c", line 4: PARALLELIZED, user pragma used
"auto.c", line 7: not parallelized, loop inside OpenMP region
$ er_src -scc parallel auto.o
```

```
   Source OpenMP region below has tag R1
    Variables autoscoped as SHARED in R1: b, c, a, m, n
    Variables autoscoped as PRIVATE in R1: sum, j
    Private variables in R1: j, sum, i
    Shared variables in R1: n, b, c, a, m

     3.    #pragma omp parallel for default(__auto)

   L1 parallelized by explicit user directive
   L1 parallel loop-body code placed in function _$d1A3.m1_mxv along
   with 1 inner loops
     4.    for (int i=0; i<m; i++)
     5.    {
     6.      double sum = 0.0;

   L2 not parallelized because it is inside OpenMP region R1
     7.      for (int j=0; j<n; j++)
     8.        sum += b[i][j]*c[j];
     9.      a[i] = sum;
    10.    } // End of parallel for
    11. }
```

# *The Thread Analyzer*

# An example of a data race/1

```
#pragma omp parallel default(none) private(i,k,s) \
        shared(n,m,a,b,c,d,dr)
{
   #pragma omp for
   for (i=0; i<m; i++)
   {
     int max_val = 0;

     s = 0 ;
     for (k=0; k<i; k++)
        s += a[k]*b[k];
     c[i] = s;

     dr = c[i];
     c[i] = 3*s - c[i];
     if (max_val < c[i]) max_val = c[i];
     d[i] = c[i] - dr;
   }
} /*-- End of parallel region --*/
```

**Where is the data race ?**

# An example of a data race/2

```
#pragma omp parallel default(none) private(i,k,s) \
        shared(n,m,a,b,c,d,dr)
{
   #pragma omp for
   for (i=0; i<m; i++)
   {
```

**Here is the data race !**

```
% cc -xopenmp -fast -xvpara -xloopinfo -c data-race.c
"data-race.c", line 9: Warning: inappropriate scoping
        variable 'dr' may be scoped inappropriately
        as 'shared'
        . read at line 24 and write at line 21 may
          cause data race
```

```
     dr = c[i];
     c[i] = 3*s - c[i];
     if (max_val < c[i]) max_val = c[i];
     d[i] = c[i] - dr;
   }
} /*-- End of parallel region --*/
```

# A True Story

- *SPECOMP Benchmark fma3d*
- *101 source files; 61,000 lines of Fortran code*
- *Data race in platq.f90 caused sporadic core dumps*
- *It took several engineers and 6 weeks of work to find the data race manually*

# Subroutine executed in parallel

```
      < 1927 lines omitted >
    SUBROUTINE PLATQ_STRESS_INTEGRATION ( NEL,SecID,MatID )
      < 45 lines omitted >
!!OMP THREADPRIVATE (/PLATQ_COMMON/)
!!
      < 7 lines omitted >
      LOGICAL, SAVE :: FIRST = .TRUE.
      < 17 lines omitted >
!! Define constants.
!!
      IF (FIRST) THEN
        SQRT6o1 = SQRT (6.0D+0/1.0D+0)
        SQRT5o6 = SQRT (5.0D+0/6.0D+0)
        FIRST = .FALSE.
      ENDIF
!!
!! Compute current element thickness based on constant volume.
!!      Thickness = SECTION_2D(SecID)%Thickness *
     &  PLATQ(NEL)%PAR%Area / PLATQ(NEL)%RES%Area
       < 425 lines omitted >
```

# Subroutine executed in parallel

```
      < 1927 lines omitted >
     SUBROUTINE PLATQ_STRESS_INTEGRATION ( NEL,SecID,MatID )
      < 45 lines omitted >
!!OMP THREADPRIVATE (/PLATQ_COMMON/)
!!
      < 7 lines omitted >
       LOGICAL, SAVE :: FIRST = .TRUE.
      < 17 lines omitted >
!! Define constants.
!!
       IF (FIRST) THEN
         SQRT6o1 = SQRT (6.0D+0/1.0D+0)
         SQRT5o6 = SQRT (5.0D+0/6.0D+0)
         FIRST = .FALSE.
       ENDIF
!!
!! Compute current element thickness based on constant volume.
!!       Thickness = SECTION_2D(SecID)%Thickness *
      &  PLATQ(NEL)%PAR%Area / PLATQ(NEL)%RES%Area
       < 425 lines omitted >
```

shared

Data Race!

Compiler Re-orders

# Run-time behavior

Time

## Thread 1

```
FIRST = .FALSE
```

**<Context Switch>**

```
SQRT6o1 = SQRT (6.0D+0/1.0D+0)
SQRT5o6 = SQRT (5.0D+0/6.0D+0)

<use SQRT6o1 and SQRT5o6>
```

## Thread 2

```
IF (FIRST) THEN
  FIRST = .FALSE.
  SQRT6o1 = SQRT (6.0D+0/1.0D+0)
  SQRT5o6 = SQRT (5.0D+0/6.0D+0)
ENDIF

<use SQRT6o1 and SQRT5o6>
```

***Quiz: Possible Solution ?***

# A possible eff cient solution

```
     < 1927 lines omitted >
    SUBROUTINE PLATQ_STRESS_INTEGRATION ( NEL,SecID,MatID )
     < 45 lines omitted >
!!OMP THREADPRIVATE (/PLATQ_COMMON/)
!!
     < 7 lines omitted >
      LOGICAL, SAVE :: FIRST = .TRUE.
     < 17 lines omitted >
!! Define constants.
!!
!$omp single
      IF (FIRST) THEN
        SQRT6o1 = SQRT (6.0D+0/1.0D+0)
        SQRT5o6 = SQRT (5.0D+0/6.0D+0)
        FIRST = .FALSE.
      ENDIF
!$omp end single
!!
!! Compute current element thickness based on constant volume.
!!      Thickness = SECTION_2D(SecID)%Thickness *
     &  PLATQ(NEL)%PAR%Area / PLATQ(NEL)%RES%Area
      < 425 lines omitted >
```

barrier included

# Bottom line about Data Races

***Data Races Are Easy To Put In***
***But***
***Very Hard To Find***

***That is why a special tool to find data races is a "must have"***

# The Thread Analyzer

- ***Detects threading errors in a multi-threaded program:***
  - ***Data race and/or deadlock detection***
- ***Parallel Programming Models supported*:***
  - ***OpenMP***
  - ***POSIX Threads***
  - ***Solaris Threads***
- ***Platforms: Solaris on SPARC, Solaris/Linux on x86/x64***
- ***Languages: C, C++, Fortran***
- ***API provided to inform Thread Analyzer of user-def ned synchronizations***
  - ***Reduce the number of false positive data races reported***

****) Legacy Sun and Cray parallel directives are supported too***

# About The Thread Analyzer

- ***Getting Started:***

  ***http://developers.sun.com/sunstudio/downloads/ssx/tha/tha_getting_started.html***

- ***Provide feedback and ask questions on the Oracle Solaris Studio Tools Forum***

  ***http://developers.sun.com/sunstudio/community/forums/index.jsp***

# Using The Thread Analyzer

1. *Instrument the code*

```
% cc -xinstrument=datarace source.c
```

2. *Run the resulting executable under the collect command*. At runtime, memory accesses and thread synchronizations will be monitored. Any data races found will be recorded into a log f le*

```
% collect -r [ race | deadlock ] a.out
```

2. *Display the results:*

```
% er_print [-races | -deadlock] tha.1.er
```

*(Command-line interface)*

```
% tha tha.1.er
```

*(Customized Analyzer GUI)*

***) Executable will run slower because of instrumentation**

# Support for deadlock detection

- *The Thread Analyzer can detect both potential deadlocks and actual deadlocks*
- *A potential deadlock is a deadlock that did not occur in a given run, but can occur in different runs of the program depending on the timings of the requests for the locks by the threads*
- *An actual deadlock is one that actually occurred in a given run of the program*
  - *An actual deadlock causes the threads involved to hang, but may or may not cause the whole process to hang*

# Example of a Data Race

```
#pragma omp parallel shared(n)
   {
#pragma omp single
     {printf("Number of threads: %d\n",omp_get_num_threads());}

     n = omp_get_thread_num();

     printf("Hello Data Race World n = %d\n",n);
   } /*-- End of parallel region --*/
```

***The output is correct:***

```
Number of threads: 4
Hello Data Race World n = 3
Hello Data Race World n = 2
Hello Data Race World n = 1
Hello Data Race World n = 0
```

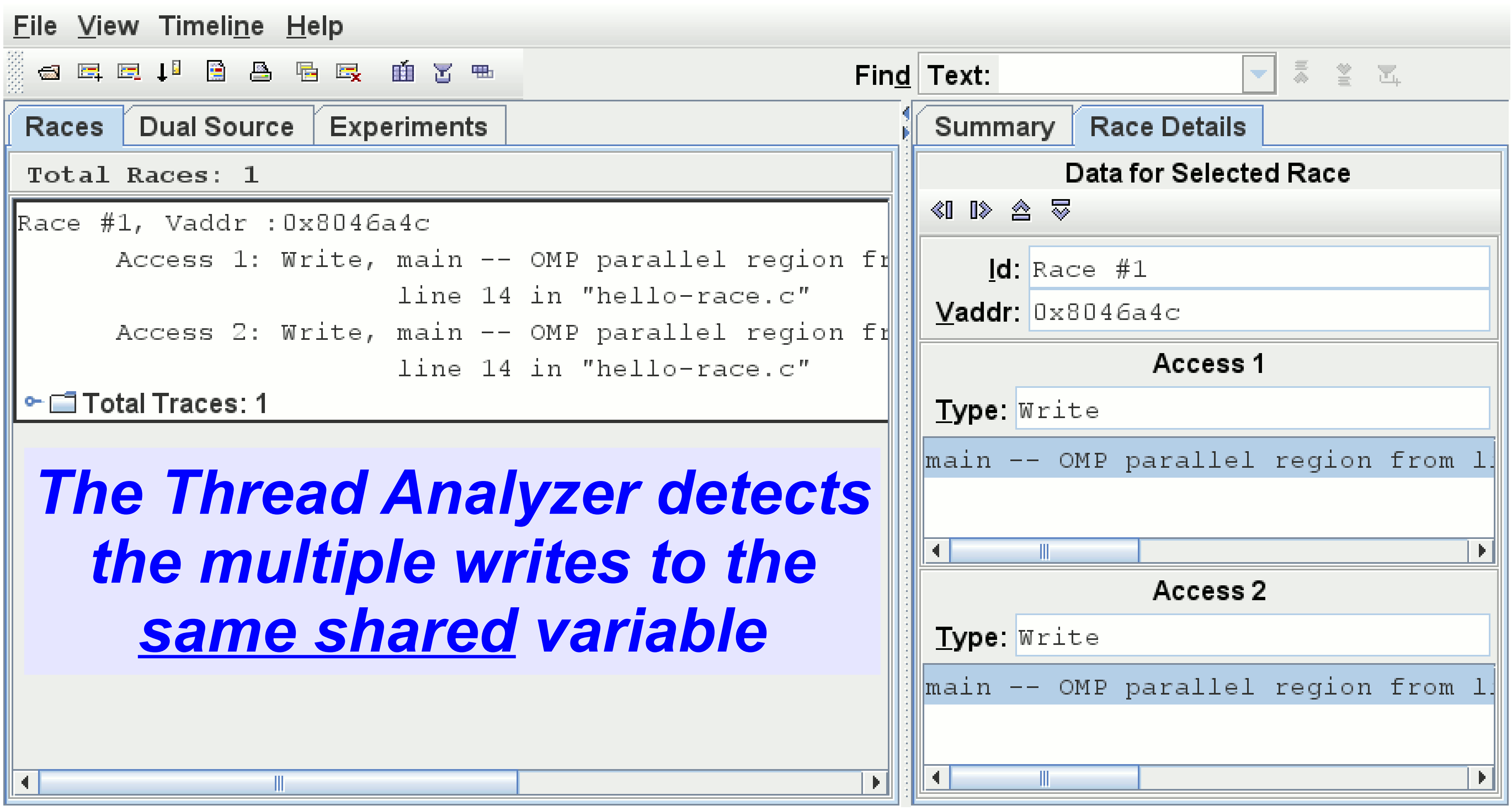
***Let's see what the Thread Analyzer says:***

# Example command line output

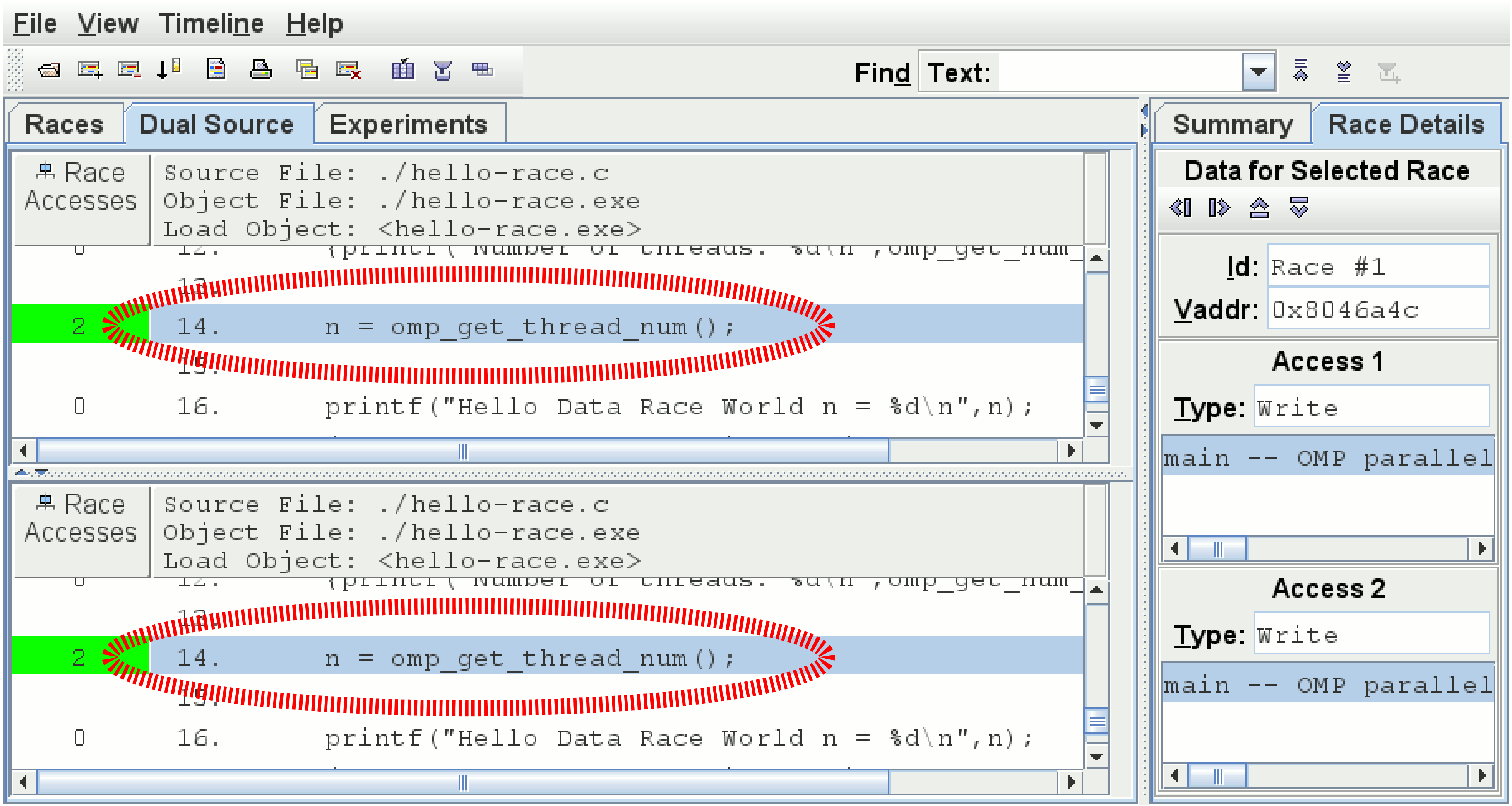
```
Total Races:  1 Experiment:  race.er

Race #1, Vaddr: 0x8046a4c
      Access 1: Write, main -- OMP parallel region from
                line 9 [_$p1A9.main] + 0x000000B9,
                       line 14 in "hello-race.c"
      Access 2: Write, main -- OMP parallel region from
                       line 9 [_$p1A9.main] + 0x000000B9,
                       line 14 in "hello-race.c"
  Total Traces: 1
```

***The Thread Analyzer detects the multiple writes to the same shared variable***

# Thread Analyzer GUI - Races



# Thread Analyzer GUI - Sources




***The source lines of the conflicting writes are shown in the "Dual Source" tab***

# Revisiting the True Story

- ***SPECOMP Benchmark fma3d***
- ***101 source f les; 61,000 lines of Fortran code***
- ***Data race in platq.f90 caused sporadic core dumps***
- ***It took several engineers and 6 weeks of work to f nd the data race manually***

***With the Oracle Solaris Studio Thread Analyzer, the data race was detected in just a few hours!***

# Avoiding Data Races

- ***Rule #1 - Avoid a simultaneous update of shared data***
- ***Rule #2 - Make sure the data sharing attributes (e.g. private, shared, etc) are correct***
  - ***Consider using Sun's autoscoping to assist you***
- ***Rule #3 - Use the Oracle Solaris Studio Thread Analyzer***
- ***OpenMP provides several constructs to help:***
  - ***Critical Section - Only one thread at a time***
  - ***Explicit Barrier - All threads wait for the last one***
  - ***Atomic Construct - Lightweight critical section***
  - ***Single Region - Only one thread; has implied barrier***

# Summary OpenMP

- ***OpenMP provides for a small, but yet powerful, programming model***
- ***It can be used on a shared memory system of any size***
  - ***This includes a single socket multicore system***
- ***Compilers with OpenMP support are widely available***
- ***The tasking concept opens up opportunities to parallelize a wider range of applications***
- ***Oracle Solaris Studio has extensive support for OpenMP developers***